Architecture logicielle générique et approche à base de modèles pour la sûreté de fonctionnement des systèmes interactifs critiques

Fayollas Camille

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue par :
Camille Fayollas
le mardi 21 juillet 2015

Titre :

Architecture logicielle générique et approche à base de modèles pour la sûreté de fonctionnement des systèmes interactifs critiques

---

École doctorale et discipline ou spécialité :
ED MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de recherche :
IRIT et LAAS-CNRS

Directeur/trice(s) de Thèse :

Philippe Palanque et Jean-Charles Fabre

Jury :
Nicolas Roussel, Directeur de Recherche, INRIA Lille (Rapporteur)
François Taïani, Professeur, Université de Rennes 1 (Rapporteur)
Yannick Déléris, Ingénieur de recherche, Airbus Opérations (Examinateur)
Hervé Girod, Ingénieur de recherche, Dassault Aviation (Examinateur)
David Navarre, Maître de conférences, Université Toulouse 1 (Examinateur)
Juan Carlos Ruiz, Professeur, Université Polytechnique de Valencia (Examinateur)
Jean-Charles Fabre, Professeur, INPT (Co-directeur)
Philippe Palanque, Professeur, Université Toulouse 3 (Directeur)

REMERCIEMENTS

Les travaux présentés dans ce mémoire ont bénéficié d'une étroite collaboration entre le département Cockpit d'Airbus Operations, l'équipe Interactive Critical Systems de l'Institut de Recherche en Informatique de Toulouse et le groupe Tolérance aux fautes et Sûreté de Fonctionnement du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique. Je tiens à remercier tous les acteurs qui ont rendu possibles ces recherches.

Merci à François Taïani et Nicolas Roussel pour m'avoir fait l'honneur d'accepter de rapporter ma thèse et m'avoir autorisée à soutenir mes travaux de thèse Ils ont su prendre le temps d'étudier et de commenter ce manuscrit ce qui m'a permis de l'améliorer considérablement. Merci également à Hervé Girod et Juan Carlos Ruiz pour avoir accepté de se déplacer pour faire partie de mon jury et ainsi partager avec moi leur avis sur mes travaux.

La préparation de cette thèse a été rendue possible grâce au financement d'Airbus Operations. Ainsi, je remercie particulièrement Yannick Déléris, Philippe Palanque et Jean-Charles Fabre pour avoir monté ce projet et pour la confiance qu'ils m'ont accordée en me confiant ce sujet de recherche.

Je tiens également à exprimer un grand merci à mes encadrants académiques et industriels.

Merci à Philippe Palanque pour m'avoir accordé sa confiance lors de mon stage de fin d'étude en m'impliquant rapidement dans les problématiques de l'équipe, pour m'avoir fait (re)découvrir le monde de la recherche et découvrir le domaine de l'interaction homme-machine que je ne connaissais pas du tout. Merci pour sa patience et les nombreuses répétitions des définitions et principes de base de ce domaine que j'ai ainsi pu finir par retenir. Merci pour son accueil, son énergie et son ouverture d'esprit.

Merci à Jean-Charles Fabre qui a su me donner le goût et l'envie de faire de la recherche à travers ses enseignements et la découverte du domaine de la sûreté de fonctionnement informatique et de la sécurité. Merci pour sa confiance professionnelle qui m'a permis de découvrir le monde de la recherche à travers les différents stages et projets dans lesquels il m'a donné la possibilité de m'impliquer. Merci pour les coups de pieds dans la fourmilière quand ils s'imposaient et qui ont permis de faire avancer les choses.

Merci pour leurs apports scientifiques, pour l'ouverture à l'international qu'ils ont permis de donner à ces travaux, le travail collaboratif malgré les incompréhensions dues aux différences de domaine de recherche et pour leur passion de la recherche hautement communicative. Merci de rester une source d'inspiration et une référence malgré ces quatre années pendant lesquelles j'ai pu évoluer et apprendre à me faire ma propre opinion.

Merci également à David Navarre, grand oublié des papiers officiels pour son encadrement et ses conseils techniques. Merci pour nos longues discussions techniques sur les différentes modélisation comportementales réalisées dans cette thèse, elles ont permis d'améliorer grandement la qualité des travaux présentés. Merci également pour avoir su endosser le rôle du directeur de thèse en répondant par exemple à mes questions existentielles sur la rédaction du manuscrit lorsque Philippe était en vadrouille.

Merci à Yannick Déléris pour ses conseils, son suivi régulier, son point de vue extérieur et industriel qui ont permis de donner à ces travaux une application industrielle et concrète. Merci pour sa patience et sa disponibilité pour les échéances scientifiques telles que la rédaction du manuscrit de thèse et les divers articles de recherche.

Merci à toute l'équipe ICS pour les nombreux échanges, l'accueil, le soutien et les nombreux encouragements. Un merci tout particulier à mes deux voisines de bureau, les deux petites vieilles. Merci à Célia Martinie pour m'avoir encouragée et impliquée dans des travaux sortant parfois légèrement du cadre de la thèse. Merci pour tous ses conseils qu'elle a su me donner durant ces 4 années, elle reste un modèle dont j'espère pouvoir un jour suivre les traces. Merci à Martina Ragosta pour les séances de piscine et les nombreux échanges, scientifiques ou non, pour m'avoir fait découvrir la gastronomie italienne et pour avoir partagé ses recettes. Merci à Éric Barboni (dit Didier) pour sa patience lors de ma découverte de l'outil PetShop et de la modélisation ICO, merci pour son aide scientifique sans laquelle aucun de mes travaux ne fonctionnerait à l'heure actuelle. Merci également pour l'organisation des diverses sorties sportives. Merci à Marco Winckler pour son énergie, sa connaissance et sa passion des orchidées ainsi que des dédales administratifs. Merci également pour les petits défis quotidiens, les blagues brésiliennes et l'implication dans l'organisation de divers événements scientifiques. Merci à tous les autres membres de l'équipe, actuels et anciens pour leurs conseils et encouragements et plus particulièrement : Regina Bernhaupt, Mickael Pirker, Adrienne Tankeu-Choitat, Martin Cronel, Racim Fahssi, François Manciet, Antoine Desnos, Bastien Gatellier, Raphaël Guenon, Romuald Deshayes, Thiago Rocha-Silva, Caio Stein D'Agostini, Jose Da Silva, Thomas Mirlacher …

Un merci tout particulier à Arnaud Hamon, compagnon de galère privilégié. Merci pour nos débats scientifiques, pour nos discussions moins scientifiques. Merci pour son soutien et son amitié et pour toutes ces soirées passées ensemble, à travailler ou à s'amuser !

Merci à tout le groupe TSF, aux permanents car ils contribuent pour beaucoup à la bonne entente générale dans l'équipe TSF, un merci particulier à Michaël Lauer, Matthieu Roy, Nicolas Rivière, Karama Kanoun, Mohamed Kaaniche, Yves Crouzet, Agnan de Bonneval, … .Merci également aux doctorants, docteurs, post-doctorants et stagiaires que j'ai pu rencontrer et dont je me suis liée d'amitié avec nombre d'entre eux. Il est difficile de tous les citer mais je veux essayer : Yann, Mathilde, Pierre, Thibaut, Roberto, Ludovic, Ivan, Hélène, Joris, Miruna, Carla, Amira, Quynh Anh, Moussa, Benoît, Julien, Guthemberg, Anthony, Jesus, Robert, Kossi, Jimmy et tous ceux que j'oublie. Enfin, un merci tout particulier à Kalou pour ses nombreuses relectures et sa générosité en encre rouge qui m'a permis d'améliorer ce manuscrit.

Merci à Adrienne Tankeu-Choitat pour m'avoir encadré durant mon stage de fin d'étude et pour m'avoir initiée à ce sujet de recherche qui me passionne.

Merci à Gabriel Bustamante, Rémy Palustran et plus particulièrement Martin Cronel pour le travail fourni sur le simulateur ARISSIM.

Merci à l'équipe Display system d'Airbus, plus particulièrement à Antonio Pons et Sofyan Su.

Merci à Jacqueline pour sa bonne humeur et ses anecdotes régulières et merci à tous les membres de l'équipe SIERA pour animer les couloirs vides du 1er étage du 1R1.

Merci à tous mes amis, qui m'ont soutenu pendant ces quatre années malgré la distance ou les autres éloignements de la vie. Merci à Mohamed pour nos longues conversations téléphoniques. Merci à Myriam et Adrien, Eva et Tristan pour leur joie de vivre, leur amitié et l'exemple qu'ils prodiguent. Merci à Manuel pour nos retrouvailles toujours enrichissantes. Merci à Nicolas pour ces soirées de travail et de répétitions qui m'ont beaucoup aidée ainsi que pour toutes ses réflexions sur la vie avec lesquelles je n'adhère pas forcément mais qui sont restées toujours aussi passionnantes … Merci à

Marjorie et Éric pour leur soutien constant et toutes ces soirées qui m'ont permis de tenir le cap dans les moments difficiles. Merci à Alexandre d'avoir su rester et de rester le dude. Merci à Rémy et Julien pour leur amitié et leurs blagues. Merci à Gaëlle et David pour les nombreuses soirées et leur soutient. Merci à Marine pour son amitié à toute épreuve. Merci à Sarah, Louise, Julien, Marie-Bénédicte et Guillaume pour leur amitié et pour avoir su insister quand j'avais besoin de sortir alors que je le refusais. Enfin, merci à tous les autres que j'ai pu oublier.

Enfin, merci à toute ma famille, pour leur soutien sans faille, l'aide et la confiance qu'ils ont su me donner. Merci à Monique Lucat, ma marraine si attentive et à son mari Bernard, aux Dulac et aux Veyrac, merci à Nathalie et Béatrice Mahé, merci à mes grands-parents et merci aux cousines Mahé Isabelle, Stéphanie et Émilie, merci aux cousins Roussel, Rémi, Sophie, Lucile et Élise.

Un merci tout particulier à mes parents, Jean-Philippe et Bénédicte pour leur soutient continu et les nombreuses relectures et l'amour qu'ils me prodiguent chaque jour. Merci à mes frère et sœurs de m'accompagner au quotidien. Chantal pour m'avoir supportée (dans les deux sens du terme) tout au long de cette dernière année de thèse pas toujours évidente, merci pour son énergie débordante. Florent pour son calme et son ordre, merci pour ses interrogations qui m'émerveillent tous les jours et nos discussions scientifiques. Et enfin Cécile pour sa joie de vivre indéfectible, ses câlins du week-end que j'attends toujours avec impatience, merci de rester la petite dernière du haut de tes 17 ans.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

Les systèmes interactifs ont considérablement évolué durant les dernières décennies, notamment dans les domaines grand public. Ainsi, les systèmes interactifs qui ont été démocratisés, tels que les ordinateurs de bureau, sont en constante évolution et intègrent des techniques d'interactions sophistiquées telles que celles permettant l'utilisation d'un écran tactile ou de la reconnaissance vocale pour commander un système.

Il existe cependant des domaines dans lesquels l'intégration de ces évolutions reste très difficile. C'est notamment le cas des systèmes interactifs critiques que l'on retrouve par exemple dans les cockpits d'avions civils ou dans les centres de commande et contrôle des centrales nucléaires. Ces domaines commencent à peine à intégrer des systèmes interactifs semblables à ceux que nous utilisons quotidiennement. Ainsi, si nous prenons l'exemple des cockpits d'avions civils, seuls les cockpits des avions les plus récents tels que l'A380, l'A350 ou le Boeing 787, intègrent, en plus des interfaces homme-machine analogiques, l'utilisation de systèmes interactifs pour la commande et le contrôle des systèmes avioniques. Ces systèmes interactifs sont composés d'écrans permettant l'affichage d'applications avec lesquelles le pilote peut interagir à travers l'utilisation d'une souris et d'un clavier.

L'utilisation des systèmes interactifs dans les cockpits d'avions est régie par des exigences de certification qui imposent que ceux-ci soient conçus et développés avec un niveau de sûreté de fonctionnement cohérent avec celui des systèmes avioniques qu'ils permettent de commander et contrôler. Ainsi, pour que les systèmes interactifs puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques, il est nécessaire qu'ils soient conçus et développés avec un niveau de sûreté de fonctionnement suffisant. Ceci impose l'utilisation d'approches de prévention des fautes et de tolérance aux fautes. À l'heure actuelle, la conception et le développement des systèmes interactifs n'intègrent pas ce genre d'approche et, bien que le niveau de sûreté de fonctionnement des systèmes interactifs est suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques non critiques, il n'est pas suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques.

Par ailleurs, les systèmes interactifs sont destinés à être utilisés par des humains et il est fondamental de s'assurer qu'ils soient utilisables car, comme le suggère Susan Dray dans son slogan "*If the user can't use it, it doesn't work*", si un système n'est pas utilisable, l'utilisateur ne pourra pas s'en servir, et donc, par extension, ce système sera incapable de fournir le service pour lequel il a été conçu. Il est donc fondamental de prendre en compte cette propriété lors de la conception et du développement du système.

Hypothèses de la thèse et contributions

L'objectif de nos travaux est de démontrer qu'il est possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques.

Pour cela, nous proposons à la fois une architecture logicielle générique et une approche à base de modèles pour réaliser des systèmes interactifs sûrs de fonctionnement, ce qui permettra d'envisager de les utiliser pour la commande et le contrôle de systèmes critiques.

Premièrement, nous proposons une approche basée sur les modèles pour des systèmes interactifs zéro-défaut. L'objectif de cette approche est d'éviter et supprimer autant que possible les fautes logicielles de développement. Elle repose sur la description des différents composants logiciels d'un système interactif à l'aide d'une technique de description formelle permettant une description concise, complète et non ambigüe ainsi qu'une analyse formelle sur les modèles obtenus.

Deuxièmement, nous proposons une architecture logicielle pour des systèmes interactifs tolérants aux fautes. L'objectif de cette architecture est de permettre l'introduction de mécanismes de tolérance aux fautes dans les systèmes interactifs afin de traiter les fautes logicielles résiduelles et les fautes

naturelles. Pour cela, après avoir étudié les architectures classiques de sûreté de fonctionnement et leur applicabilité aux différents composants logiciels des systèmes interactifs, nous en choisissons une particulièrement adaptée au contexte de la thèse et développons son utilisation au sein des systèmes interactifs avioniques.

Troisièmement, nous proposons des outils ainsi qu'une architecture matérielle pour la mise en œuvre de notre approche. Nous proposons également des pistes de validation permettant notamment de garantir le maintien de l'utilisabilité du système malgré l'ajout des mécanismes de tolérance aux fautes.

Enfin, nous étudions l'application de notre approche au travers une étude de cas issue du milieu de l'avionique : une version logicielle du Flight Control Unit, système permettant notamment aux pilotes de contrôler le pilote automatique de l'avion.

Structure du mémoire

Ce mémoire est structuré en trois parties qui nous permettent de présenter le contexte et l'état de l'art, les contributions et enfin l'application de notre approche à une étude de cas.

La première partie présente le contexte de la thèse ainsi que les travaux existants pertinents par rapport à ce contexte.

Le Chapitre 1 présente, à travers l'étude de l'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils, la problématique à laquelle nous nous efforçons de répondre dans cette thèse : est-il possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle des systèmes critiques dans les cockpits d'avions civils ?

Le Chapitre 2 présente les différentes définitions nécessaires à la compréhension de ce document ainsi que les travaux existants concernant la sûreté de fonctionnement des systèmes interactifs.

Le Chapitre 3 présente le contexte applicatif de la thèse : les systèmes interactifs présents dans les cockpits d'avions civils, leur architecture matérielle et logicielle ainsi que les contraintes que l'on doit respecter pour leur développement.

La deuxième partie présente l'approche globale que nous proposons pour concevoir et développer des systèmes interactifs sûrs de fonctionnement. Celle-ci est composée d'une approche à base de modèles visant à réaliser des systèmes interactifs zéro-défaut et d'une architecture logicielle générique pour des systèmes interactifs tolérants aux fautes.

Le Chapitre 4 définit le périmètre des travaux que nous présentons en détaillant les hypothèses que nous avons faites, les modes de défaillances que nous cherchons à éviter ainsi que le modèle de faute regroupant les fautes que nous pouvons traiter grâce à l'utilisation de notre approche. Nous définissons ensuite une architecture générique, fondée sur le modèle architectural ARCH pour les systèmes interactifs afin d'en détailler tous les composants logiciels. Pour finir, nous présentons les principes généraux de l'approche globale.

Le Chapitre 5 définit une approche à base de modèles pour la conception et le développement de systèmes interactifs zéro-défaut. Plus concrètement, nous proposons de modéliser tous les composants logiciels des systèmes interactifs à l'aide d'une notation formelle nommée ICO.

Le Chapitre 6 définit une architecture logicielle générique et tolérante aux fautes pour les systèmes interactifs. Cette architecture est fondée sur l'application du principe de la programmation n-autotestable à tous les composants logiciels des systèmes interactifs.

Le Chapitre 7 présente des moyens pour la mise en œuvre de notre approche pour des systèmes sûrs de fonctionnement. Nous proposons premièrement l'utilisation d'un outil pour la modélisation des composants logiciels avec la notation formelle ICO. Deuxièmement, nous proposons une mise en œuvre

pour notre architecture tolérante aux fautes, fondée sur l'utilisation de l'outil pour la modélisation et d'un simulateur de système d'exploitation avionique. Enfin, nous proposons des pistes pour la validation de notre approche.

La troisième partie présente la validation de la faisabilité de notre approche. Celle-ci est effectuée à travers l'application de nos travaux à une étude de cas réaliste issue du milieu de l'avionique : un système interactif inspiré du Flight Control Unit, le système permettant de commander et contrôler le pilote automatique de l'avion et ses différents paramètres.

Le Chapitre 8 est consacré à la présentation de l'étude de cas. Il présente ainsi le système de commande et contrôle Flight Control Unit (FCU) ainsi que l'application interactive (appelée FCUS) que nous avons créée en nous inspirant de celui-ci.

Le Chapitre 9 présente la mise en œuvre de l'approche à base de modèles pour la conception et le développement zéro-défaut de cette étude de cas. Il présente ainsi la modélisation des différents composants de notre application FCUS grâce à la notation ICO.

Le Chapitre 10 présente la mise en œuvre de l'architecture logicielle tolérante aux fautes à cette étude de cas. Il instancie ainsi l'architecture logicielle tolérante aux fautes sur notre application FCUS grâce à la création d'un composant effectuant le contrôle de certaines assertions et à l'implantation de l'application tolérante aux fautes sur un simulateur de système d'exploitation avionique.

Enfin, la conclusion de ce mémoire résume les apports de cette thèse en identifiant les limitations de nos travaux et en présentant des perspectives à court et long terme pour les prendre en compte.

Partie 1. ÉTAT DE L'ART ET CONTEXTE

Contenu de la partie

Cette première partie présente le contexte de la thèse ainsi que les travaux existants pertinents par rapport à ce contexte.

Le Chapitre 1 présente, à travers l'étude de l'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils, la problématique à laquelle nous nous efforçons de répondre dans cette thèse : est-il possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle des systèmes critiques dans les cockpits d'avions civils ?

Le Chapitre 2 présente les différentes définitions nécessaires à la compréhension de ce document ainsi que les travaux existants concernant la sûreté de fonctionnement des systèmes interactifs.

Le Chapitre 3 présente le contexte applicatif de la thèse : les systèmes interactifs présents dans les cockpits d'avions civils, leur architecture matérielle et logicielle ainsi que les contraintes que l'on doit respecter pour leur développement.

Sommaire

Chapitre 1. Problématique

Sommaire

Ce chapitre présente, au travers l'étude de l'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils, la problématique à l'origine de cette thèse : est-il possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques ?

Dans la première section nous présentons l'évolution des systèmes de commande et contrôle des cockpits depuis les systèmes analogiques utilisés dans les A300 jusqu'aux systèmes interactifs utilisés dans les avions modernes tels que l'A380.

Dans la deuxième section nous présentons les besoins de sûreté de fonctionnement des systèmes de commande et contrôle dans les cockpits avioniques. Ces besoins justifient la lenteur de l'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils par rapport à celle des systèmes grand public.

Dans la troisième section nous présentons en quoi l'utilisation des systèmes interactifs permet d'améliorer la performance des pilotes ainsi que de diminuer les coûts de fabrication de et maintenance du cockpit. Ces raisons justifient l'utilité de ces systèmes dans les cockpits d'avions civils.

Enfin, la quatrième section présente la problématique à laquelle nous nous intéressons.

1.1 L'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils

Pour faire voler l'avion, le *pilote* opère les *systèmes avioniques* grâce à des interfaces homme-machine appelées *systèmes de commande et contrôle* et situées dans le cockpit. Le terme système avionique est ici utilisé au sens large du terme, ce qui inclut tous les systèmes physiques (moteurs, capteurs, volets, …), électriques, électroniques et informatiques (pilote automatique, commandes de vol numériques, …), embarqués dans l'avion. Le terme de commande et contrôle est utilisé ici car ceux-ci servent d'intermédiaire entre un humain et des systèmes physiques. Comme le montre la Figure 1.1, c'est donc au travers de ces systèmes de commande et contrôle que le pilote peut déclencher des commandes sur les systèmes avioniques et qu'il peut contrôler (monitorer) leur état.

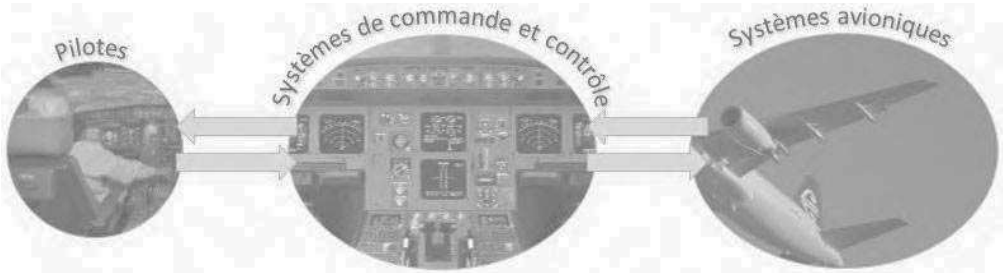


Figure 1.1. Interactions entre les pilotes et les différents systèmes de l'avion

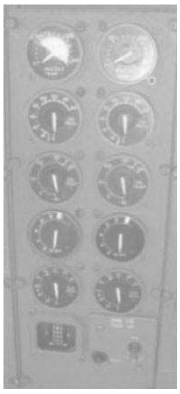
Dans les premiers cockpits d'avions civils, chaque système avionique est relié physiquement à son propre système de commande et contrôle. Ceux-ci sont principalement constitués d'instruments électromécaniques. Le contrôle de l'état des différents paramètres des systèmes avioniques se fait au travers de cadrans mélangeant des chiffres, symboles et aiguilles. La commande des systèmes s'effectue au travers de boutons physiques tels que des boutons poussoirs, des rotateurs, des commutateurs… Ces cockpits sont appelés des *cockpits analogiques* et sont typiquement constitués de plus d'une centaine d'instruments d'affichage et de commande. Le cockpit de l'A300 est une très bonne illustration de ces cockpits. La Figure 1.2 nous permet de remarquer le nombre important de boutons et de cadrans qui le constituent.

Figure 1.2. Cockpit d'un A300

Dans les années 1980, l'augmentation du nombre de systèmes avioniques implique une augmentation du nombre de systèmes de commande et contrôle et donc du nombre de cadrans et boutons présents dans les cockpits. Pour éviter cette augmentation du nombre d'affichages et de boutons, l'industrie avionique a développé, à la fin des années 1980, de nouveaux cockpits appelés les *glass cockpits*. La différence principale avec les cockpits analogiques réside dans l'introduction d'écrans permettant des affichages numériques. Ceux-ci rendent possible la juxtaposition d'affichages d'informations provenant de plusieurs systèmes avioniques sur un même écran. Le contrôle des informations provenant des systèmes avioniques est donc grandement facilité pour les pilotes. Cependant, pour commander les systèmes avioniques, les pilotes utilisent, comme dans les cockpits analogiques, des boutons physiques mis à disposition à côté des affichages. Cette génération de cockpits concerne toute la famille des A320, A330 et A340 mais également l'A300-600 ainsi que les Boeing 737, 757 et 767. La Figure 1.3 représente un cockpit d'A320 sur lequel on peut voir que les six écrans remplacent un grand nombre de cadrans qui étaient présent dans le cockpit de l'A300. Cette génération de cockpit a notamment permis la réduction de l'équipage naviguant de trois à deux personnes en participant à la suppression du mécanicien de bord.

Figure 1.3. Cockpit d'un A320

Pour illustrer cette première évolution des systèmes de commande et contrôle dans les cockpits, nous prenons l'exemple de l'affichage des paramètres moteurs. La Figure 1.4 montre l'évolution de cet affichage entre les cockpits analogiques et les glass cockpits. Nous pouvons ainsi constater que les huit cadrans à aiguille présents dans le cockpit analogique de l'A300 (Figure 1.4-a) sont remplacés, dans le glass cockpit de l'A320 (Figure 1.4-b), par un affichage numérique permettant la juxtaposition de l'affichage de ces informations sur un seul écran.



a) Affichage des paramètres moteurs d'un A300

b) Affichage des paramètres moteurs d'un A320

Figure 1.4. Évolution de l'affichage des paramètres moteurs entre les cockpits analogiques (a) et les glass cockpits (b)

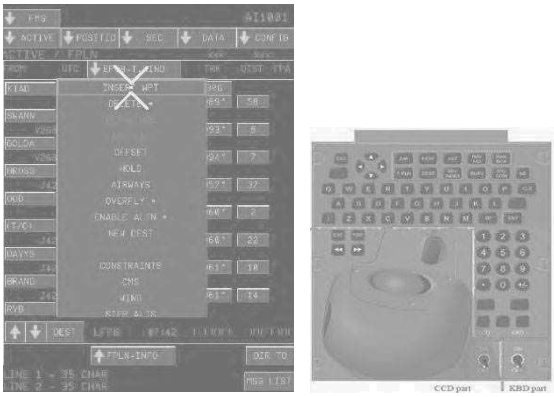
Toujours dans un souci de faire face à l'augmentation du nombre de systèmes de commande et contrôle, l'industrie avionique, a développé, au début des années 2000, le standard ARINC 661 (AEEC 2013), toujours en évolution à l'heure actuelle, qui permet la création des *cockpits interactifs*. L'innovation majeure de ces cockpits est l'introduction d'applications interactives dans le cockpit de l'avion. Ces applications interactives s'affichent sur les écrans et les pilotes peuvent interagir avec elles grâce à l'utilisation d'un dispositif combinant un clavier et une trackball, nommé KCCU (pour Keyboard and Cursor Control Unit, illustré en Figure 1.6-b). Elles sont similaires à celles que nous pouvons trouver sur nos ordinateurs de bureau où l'interaction se fait au travers d'objets graphiques. La commande des systèmes avioniques s'effectue grâce aux interactions avec les objets graphiques interactifs à l'aide d'un manipulateur de souris (ou curseur graphique). Le contrôle de l'état de ces systèmes s'effectue grâce au contrôle de l'affichage de ces objets graphiques. Les cockpits interactifs ont été intégrés pour la première fois, en 2005, dans le cockpit de l'A380 (constitué de six écrans et deux KCCU) qui est présenté en Figure 1.5.

Figure 1.5. Cockpit d'un A380

Pour illustrer cette évolution, nous prenons l'exemple du système de commande et contrôle du système de gestion de vol appelé FMS (pour Flight Management System) qui permet de gérer le plan de vol, calculer la trajectoire et la position de l'avion… La Figure 1.6 montre l'évolution du système de commande et contrôle du FMS entre les glass cockpits et les cockpits interactifs. Le système de commande et contrôle du FMS dans l'A320 (Figure 1.6-a) est appelé MCDU (pour Multipurpose Control and Display Unit). Il composé d'une zone d'affichage et de boutons poussoirs l'entourant qui

permettent au pilote de commander le FMS. Le système de commande et contrôle du FMS dans l'A380 (Figure 1.6-b) est une application interactive appelée MFD (pour Multi Function Display). Les pilotes peuvent interagir avec l'application affichée sur l'écran en utilisant le KCCU qui permet de contrôler le manipulateur de souris que l'on peut voir sur l'image (symbole jaune situé au dessus de la fonction `Insert WPT` permettant d'insérer une étape au plan de vol).

a) *Système de commande et contrôle du FMS dans l'A320 : le MCDU*



b) Système de commande et contrôle du FMS dans l'A380 : le MFD et le KCCU

Figure 1.6. Évolution du système de commande et contrôle du FMS entre les glass cockpits (a) et les cockpits interactifs (b)

Dans la suite du document, nous nous référons aux systèmes de commande et contrôle typiques des cockpits interactifs (un ou plusieurs écrans affichant des applications interactives avec lesquelles peuvent interagir les pilotes grâce à l'utilisation d'un dispositif couplant un clavier et une trackball) en tant que *systèmes interactifs*. Cette notion sous-entend en effet la notion de système *informatisé* destiné à être utilisé par des humains et inclut les systèmes grand public similaires à ceux qui nous intéressent (par exemple, les ordinateurs de bureau).

1.2 Les freins à cette évolution

La frise présentée en Figure 1.7 synthétise les principales évolutions des cockpits d'avions civils ainsi que les différents types de cockpits que nous avons définis. Il est important de noter que chaque nouvelle évolution cohabite avec les technologies précédentes. Ainsi, on trouve dans les glass cockpits un certain nombre de systèmes de commande et contrôle analogiques. De la même manière, on trouve dans les cockpits interactifs, des systèmes de commande et contrôle analogiques ainsi que, comme dans les glass cockpits, des affichages numériques associés à des boutons physiques. Cette mixité s'explique par le fait qu'elle permet de profiter à la fois des avantages des nouvelles technologies et de la sûreté de fonctionnement des systèmes de commande et contrôle traditionnels.

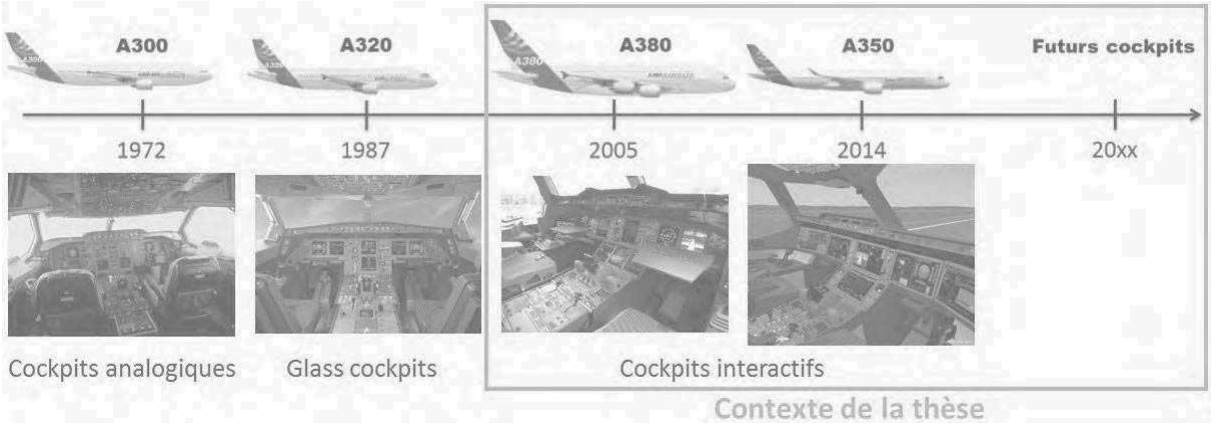


Figure 1.7. Évolution des cockpits d'avions civils

La sûreté de fonctionnement d'un système est définie comme la propriété d'un système qui permet de placer une confiance justifiée dans le service qu'il délivre (Laprie, Deswartes, et al. 1996). Dans le cas d'un avion civil opéré par une compagnie aérienne, le service fourni correspond au transport des passagers d'une ville de départ vers une ville d'arrivée. La sûreté de fonctionnement de l'avion permet donc de garantir que l'on peut placer une confiance justifiée dans le fait qu'il permettra de fournir ce service. Le contrôle du vol est prioritaire. C'est à cet objectif de sûreté de fonctionnement, permettant de garantir la sécurité des passagers, que nous nous intéressons.

La sûreté de fonctionnement de l'avion dépend de celle des systèmes avioniques qui le composent. Les systèmes avioniques qui risquent, en cas de défaillance, de mettre en danger la vie des passagers et de l'équipage de l'avion (par exemple, les moteurs) sont appelés les systèmes avioniques critiques. Ils doivent être développés avec un niveau de sûreté de fonctionnement suffisant pour garantir la sécurité des passagers. Le niveau de sûreté de fonctionnement est mesuré en termes de probabilité de défaillance par heure de vol et il doit, pour les systèmes critiques, être inférieur à 10^{-9}/h.

De la même manière que les systèmes avioniques critiques doivent être développés avec un niveau de sûreté de fonctionnement suffisant, il est nécessaire que leurs systèmes de commande et contrôle soient développés avec un niveau de sûreté de fonctionnement cohérent avec le service qu'ils fournissent. Le système de commande et contrôle d'un système avionique critique est donc par extension lui-même un système critique. Ainsi, les systèmes de commande et contrôle doivent être développés avec un niveau de sûreté de fonctionnement cohérent avec celui des systèmes qu'ils permettent de commander et contrôler.

Développer des systèmes sûrs de fonctionnement nécessite l'introduction de processus de développement et de mécanismes spécifiques. Ceux-ci sont coûteux à mettre en place et nécessitent des études longues et poussées pour arriver à un niveau de maturité suffisant pour pouvoir être utilisés pour le développement des systèmes critiques. Cette problématique explique le délai que l'on peut constater entre la démocratisation de nouveaux systèmes dans des applications grand public et leur introduction dans les avions. Les systèmes interactifs sont un exemple flagrant de ce délai : les ordinateurs de bureau comprenant une interface graphique ont été démocratisés pour une utilisation grand public au début des années 1990 et leur introduction dans les cockpits d'avions ne s'est faite qu'au début des années 2000.

Figure 1.8. Restrictions des zones interactives sur le cockpit de l'Airbus A380

Les systèmes interactifs soient développés, à l'heure actuelle, avec un niveau de sûreté de fonctionnement suffisamment élevé pour être utilisés dans les cockpits. Cependant, celui-ci n'est pas suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes critiques. La commande et le contrôle des systèmes critiques s'effectuent donc au travers l'utilisation des systèmes de commande et contrôle des cockpits analogiques et des glass cockpits qui, quand à eux, sont suffisamment sûrs de fonctionnement. Cette limitation explique que certaines zones de l'ensemble des

écrans du cockpit de l'A380 ne soient pas interactives et soient seulement des écrans d'affichage comme ceux des glass cockpits comme nous pouvons le voir en Figure 1.8.

1.3 Les besoins d'évolution des cockpits d'avions civils

Les évolutions des cockpits présentées dans la section 1.1 n'ont pas été choisies au hasard et sont nécessaires à la fois pour garantir la sécurité des passagers, par exemple en améliorant des performances de réalisation des tâches par le pilote, et pour des raisons financières et d'évolutivité. Ainsi, les systèmes de commande et contrôle des cockpits interactifs (les systèmes interactifs) apportent des avantages permettant premièrement d'améliorer la performance des pilotes et deuxièmement de diminuer les coûts pour l'avionneur et pour les compagnies aériennes. Nous détaillons ci-dessous ces avantages.

Amélioration des performances de réalisation des tâches par les pilotes

L'amélioration des performances de réalisation des tâches par les pilotes correspond parfaitement à l'un des trois facteurs de l'utilisabilité. En effet, l'utilisabilité est définie comme « *le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation spécifié* » par la norme ISO 9241 (International Standard Organization 1996). Dans le contexte des cockpits interactifs, la satisfaction des utilisateurs n'est pas une priorité. L'efficacité est un aspect fondamental car il vise à démontrer que le pilote arrivera à effectuer l'ensemble des tâches qu'il doit réaliser avec le cockpit mis à sa disposition. En effet, comme spécifié dans la norme CS-25.1302 (EASA 2014) (voir section 3.2.1 pour plus de détail), la certification des avions futurs nécessitera une prise en compte explicite des tâches des pilotes et la démonstration que le cockpit permet de les réaliser. Toutefois, nous considérons que cet aspect, lié à la sémantique du travail du pilote est en dehors du périmètre de la thèse et nous présentons dans la section 7.3.3 comment le travail réalisé dans cette thèse peut être intégré à une approche visant à étudier l'efficacité des cockpits interactifs. Enfin, l'efficience, qui correspond à la capacité à atteindre le résultat prévu avec un effort et un temps minimal. Les cockpits interactifs ont pour objectif d'avoir une efficience supérieure à celle de leurs prédécesseurs.

Au niveau du cockpit, l'efficience se rapporte à la performance avec laquelle les pilotes effectuent leurs tâches. Le pilotage d'un avion est une tâche complexe qui peut se décomposer en trois activités principales. Premièrement, le pilote doit commander et contrôler de nombreux systèmes complexes afin de faire voler l'avion et de le faire naviguer jusqu'à l'aéroport d'arrivée. Deuxièmement, le pilote a une activité de travail collaboratif et de coopération à la fois à l'intérieur du cockpit avec l'autre pilote, entre les différents avions du secteur, entre l'avion et le contrôle aérien et entre l'avion et le la compagnie aérienne. Troisièmement, le pilote doit également prendre en compte l'environnement de l'avion : les conditions météorologiques, les aéroports alentours … Toutes ces activités vont être effectuées au travers l'utilisation des systèmes du cockpit. L'interaction va donc se résumer à déclencher des commandes et percevoir les résultats de ces commandes. La théorie de l'action de Norman est parfaitement adaptée pour percevoir où se situent les difficultés que vont rencontrer les pilotes mais également les endroits où la conception du système pourrait améliorer les performances de la réalisation des tâches du pilote.

La distance articulatoire (Norman 1988) correspond à la l'effort nécessaire à l'utilisateur pour mettre en relation ce qu'il perçoit du système et comment il doit l'interpréter pour effectuer une action ou en déduire une information. La distance sémantique (Norman 1988) correspond à l'effort nécessaire à l'utilisateur pour mettre en correspondance ses buts et les actions qu'il peut effectuer sur le système ou les informations qu'il peut en tirer. Réduire ces deux distances permet de simplifier le travail de l'utilisateur et d'améliorer sa performance lorsqu'il effectue les tâches qui lui sont allouées.

Dans le cas du cockpit, la diminution de ces deux distances peut être illustrée par l'exemple de la saisie de la référence de pression atmosphérique en millimètres de mercure. Dans le cas d'un cockpit non interactif, cette saisie est effectuée à l'aide du FCU (Flight Control Unit) à l'aide d'un rotacteur physique indépendant de l'affichage de la valeur. La réalisation de cette même tâche à l'aide d'un système interactif comprenant une interface graphique est effectuée grâce à une interaction directe sur

l'objet interactif permettant à la fois l'affichage et la saisie de cette valeur. De plus, l'utilisation d'une interface graphique permet également d'assembler cet affichage (permettant également la saisie) avec d'autres affichages de données similaires, ce qui n'est pas toujours le cas dans un cockpit physique où chaque zone est attribuée à des équipements spécifiques, il n'est donc pas possible d'assembler dans ce cas, des affichages de données de systèmes différents.

Diminution des coûts et évolutivité

Les systèmes interactifs s'exécutent sur du matériel informatique (processeurs, écrans, …). Le matériel est similaire quelle que soit l'application interactive concernée et permet d'exécuter plusieurs applications interactives sur un seul matériel ou sur un seul type de matériel. Le coût du développement matériel est donc fortement diminué par l'utilisation de ces systèmes car elle implique le développement d'un seul type de support matériel pour toutes les applications interactives alors qu'il fallait développer un matériel spécifique (bloc matériel comportant des boutons, des jauges, des câblages…) pour chaque système de commande et contrôle dans les cockpits analogiques. L'utilisation des systèmes interactifs implique cependant une augmentation du coût de développement logiciel.

Les systèmes interactifs participent également à la diminution du coût de montage du cockpit. En effet, il est beaucoup moins cher de monter un réseau informatique plutôt que relier de nombreux systèmes analogiques entre eux. Ceci est d'autant plus vérifié que, pour des raisons de sûreté de fonctionnement, les câbles reliant les systèmes entre eux ainsi que les systèmes eux-mêmes sont souvent redondants : de la même manière, deux réseaux informatiques sont moins coûteux à installer que deux réseaux de câbles permettant de relier des systèmes analogiques entre eux.

La maintenance et l'évolutivité du cockpit sont également nettement moins coûteuses lorsque l'on utilise des systèmes interactifs. En effet, le rajout ou la modification d'une fonctionnalité équivaut dans ce cas à une modification du logiciel interactif alors qu'elle équivaut, pour les cockpits analogiques ou les glass cockpit, à la conception d'un nouveau matériel, un nouveau module que l'on devra intégrer dans le cockpit. De plus, il est plus facile de faire du retrofit (remplaçant de composants obsolètes ou anciens par des composants plus récents) sur des composants logiciels que sur des composants matériels. L'évolutivité d'un cockpit d'avion civil est un point fondamental du fait de la durée de vie de ces systèmes. En effet, un avion opère généralement avec des passagers pendant une vingtaine d'année et les différents programmes avioniques peuvent durer de 20 à 30 ans. Ceci implique qu'un même type d'avion peut avoir une durée de vie de plus de 50 ans. Il y a donc un besoin d'évolutivité important afin de pouvoir garantir l'évolution des cockpits d'un même type d'avion.

Ces systèmes de commande et contrôle permettent également de diminuer le nombre d'équipements de commande et contrôle dans le cockpit ainsi qu'une forte diminution de la masse et du volume de ceux-ci (les équipements matériels étant remplacés par un seul équipement intégrant du logiciel).

Enfin, du fait de la similarité entre les systèmes de commande et de contrôle des cockpits interactifs avec les ordinateurs utilisés par le grand public, le temps d'apprentissage pourrait être diminué pour les pilotes, entrainant ainsi une diminution du coût de formation des pilotes.

1.4 Problématique

La problématique à laquelle nous nous intéressons dans cette thèse est de savoir, au vu des freins aux évolutions dans les cockpits (les besoins de sûreté de fonctionnement) et des avantages des systèmes interactifs (amélioration des performances du pilote et diminution des coûts), s'il est possible d'utiliser les systèmes interactifs pour la commande et le contrôle des systèmes avioniques critiques.

C'est donc au cœur de cette problématique que s'inscrit cette thèse : est-il possible de concevoir et développer des systèmes interactifs sûrs de fonctionnement et utilisables ? Plus particulièrement, est-il possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisants pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques ? Pour cela, des approches de prévention de faute et de tolérance aux fautes dans les architectures s'imposent.

1.5 Synthèse

Nous avons présenté dans ce chapitre l'évolution des systèmes de commande et contrôle dans les cockpits d'avions civils, depuis les cockpits des premiers avions civils (comme l'A300), utilisant des systèmes analogiques, constitués de plus d'une centaine de boutons poussoirs et rotatifs, de jauges mécaniques et d'instruments électromagnétiques ; aux cockpits des avions les plus récents (comme l'A380) utilisant des systèmes interactifs similaires à des ordinateurs de bureau.

Cette évolution est similaire, avec un retard conséquent, à celle des équipements grand public. Les raisons d'un tel retard sont les besoins de sûreté de fonctionnement des systèmes utilisés dans les avions. Ce besoin de sûreté de fonctionnement explique que les systèmes interactifs ne soient utilisés à l'heure actuelle dans les cockpits que pour la commande et le contrôle de systèmes avioniques non critiques.

Cependant, ces évolutions résultent d'un besoin d'amélioration des systèmes de commande et de contrôle dans les cockpits d'avions civils. Ainsi, les systèmes interactifs permettent d'améliorer la performance des pilotes ainsi que les coûts de fabrication et de maintenance du cockpit et plus important encore, ils permettent une meilleure évolutivité du cockpit. Ces avantages justifient le fait que l'on puisse vouloir les utiliser pour la commande et le contrôle de tous les systèmes avioniques, y compris les systèmes critiques.

Tout ceci nous amène à la problématique à laquelle nous nous efforçons de répondre dans cette thèse : est-il possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques ? C'est à cette question que nous nous efforçons de répondre tout au long de ce document.

Chapitre 2. La sûreté de fonctionnement des systèmes interactifs

Sommaire

Ce chapitre présente les caractéristiques et les propriétés des systèmes interactifs et des systèmes critiques. Ceci nous permet de caractériser les systèmes faisant l'objet de cette thèse : les systèmes interactifs critiques et de mettre en valeur les problématiques liées à ces systèmes.

La première section décrit les principales caractéristiques des systèmes interactifs et des systèmes critiques afin de définir la notion de système interactif critique.

La deuxième section décrit les différentes propriétés que l'on doit garantir lors de la conception des systèmes interactifs, plus particulièrement les notions d'utilisabilité et d'User eXperience. Cette section décrit également les propriétés principales des systèmes critiques : les notions de sûreté de fonctionnement et de sécurité. Ces deux premières descriptions nous amènent à la définition des propriétés les plus importantes pour la conception des systèmes interactifs critiques.

La troisième section présente l'état de l'art concernant les méthodes, outils et techniques pour la prise en compte des différentes fautes pouvant affecter les systèmes interactifs.

Ces trois premières sections amènent des problématiques pour le développement des systèmes interactifs et des systèmes critiques. Les trois dernières sections de ce chapitre présentent l'état de l'art des méthodes utilisées pour résoudre ces problématiques. Ainsi, la quatrième section propose des architectures logicielles permettant de garantir les propriétés internes des systèmes interactifs, la cinquième section propose des notations formelles permettant de décrire le comportement des systèmes interactifs et enfin, la sixième section présente les principales techniques de tolérance aux fautes utilisées dans le domaine des systèmes critiques.

2.1 Systèmes interactifs et systèmes critiques

2.1.1 Systèmes interactifs

Un *système interactif* est un système informatisé destiné à être utilisé par des utilisateurs humains. Ce type de système réagit, durant son exécution, aux informations qui lui sont communiquées par son ou ses utilisateur(s) en produisant, toujours au cours de son exécution, une représentation de son état interne, perceptible par son ou ses utilisateurs.

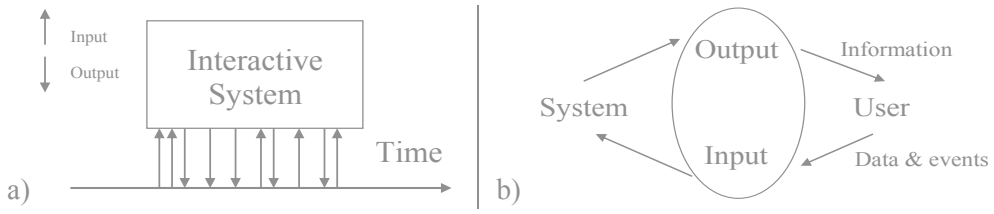


Figure 2.1. Représentations fonctionnelles d'un système interactif (Bastide et Palanque 1999)

La Figure 2.1 présente les principales caractéristiques des systèmes interactifs. La Figure 2.1-a met en évidence le fait que les entrées et les sorties du système sont entrelacées. Ceci signifie que les systèmes interactifs traitent, au cours de leur exécution, les entrées qu'ils reçoivent. Cette caractéristique les fait appartenir à la catégorie des systèmes réactifs (Pnueli 1986) qui réagissent en permanence aux sollicitations de leur environnement tout en produisant des actions sur celui-ci. Ainsi, à la différence des systèmes transformationnels qui partent d'un état initial composé de certaines données afin de les traiter et de produire des données mises à jour en sortie et où l'état initial est très important, celui-ci importe peu pour les systèmes interactifs.

La Figure 2.1-b montre que les entrées du système interactif (fournies par l'utilisateur) dépendent directement des sorties produites par le système et de la manière dont elles sont interprétées par le système. Cette figure met également en évidence le fait que les entrées du système interactif ne sont pas seulement des données mais également des événements déclenchés par l'utilisateur, provenant de périphériques tels qu'un clavier, une souris… Ainsi, les entrées des systèmes interactifs sont événementielles alors que leurs sorties sont basées sur un état, représentatif de l'état du système. Étant consommateur des sorties du système interactif et producteur de ses entrées, l'humain est placé au centre du système et chacune de ses actions sur le système doit provoquer le déclenchement d'une modification de l'état fourni en sortie par celui-ci. Du fait de la nature non prédictible du comportement d'un être humain, les entrées du système ne sont pas totalement prévisibles.

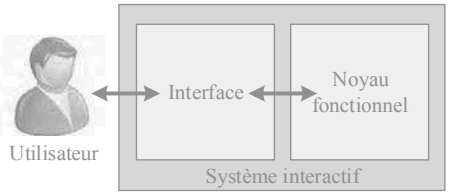


Figure 2.2. Représentation simplifiée d'un système interactif

La Figure 2.2 présente la décomposition simplifiée d'un système interactif. Celui-ci est généralement décomposé en deux parties : l'interface utilisateur et le noyau fonctionnel. L'interface utilisateur contient les éléments logiciels et matériels dédiés à la capture des entrées de l'utilisateur et à la restitution des sorties du système. Le noyau fonctionnel contient le reste du système, c'est-à-dire ses composants de calcul et de stockage de l'information (Beaudouin-Lafon 1997).

Cette représentation est très simplifiée. Elle ne permet notamment pas de mettre en évidence le très fort couplage entre le logiciel et le matériel dont dépendent les systèmes interactifs. En effet, selon les dispositifs d'entrée et de sortie, le logiciel permettant de récupérer les entrées de l'utilisateur ainsi que celui permettant d'afficher les informations à l'utilisateur peut différer grandement. Ce logiciel spécifique est appelé pilote informatique ou *driver*.

Ainsi, le développement d'un système interactif peut s'avérer très compliqué car il nécessite à la fois la prise en compte des besoins de l'utilisateur et donc l'utilisation de techniques de conception centrées sur l'utilisateur (Norman 1988) afin de garantir un système utilisable et la prise en compte de techniques d'ingénierie logicielle et matérielle afin de garantir un système qui fonctionne (Wasserman 1981). Ceci a été démontré par une étude menée par (Myers et Rosson 1992) sur un ensemble de projets en montrant qu'une moyenne de 48% du code des applications et 50% du temps de développement sont consacrés à celui de l'interface utilisateur. Afin de prendre en compte cette complexité, la communauté scientifique a développé plusieurs modèles d'architectures logicielles que nous présentons en section 2.4.

Il est également important de noter que la complexité de développement des systèmes interactifs va croissant avec les avancées technologiques. En effet, l'étude menée par (Myers et Rosson 1992) était portée sur des systèmes interactifs avec des techniques d'interaction de type *WIMP* qui signifie Window Icon Menu and Pointer (van Dam 1997). Ces techniques d'interactions permettent à l'utilisateur d'interagir à l'aide d'un ensemble de fenêtres, menus et objets graphiques manipulés avec un curseur graphique, grâce à l'utilisation d'une souris. Les techniques d'interaction plus récentes telles que les moyens de visualisation 3D, les interactions avec des écrans tactiles, l'utilisation de différents moyens d'entrée comme la voix et les gestes sont qualifiées de *post-WIMP* (van Dam 1997). Ces interactions impliquent une encore plus grande intégration entre le logiciel et le matériel : le driver d'un écran tactile sera bien évidemment plus compliqué à développer que celui d'une simple souris.

2.1.2 Systèmes critiques

Un *système critique* est généralement défini comme un système pour lequel une défaillance aura un coût supérieur à celui du développement du système. La notion de coût est à prendre ici au sens large, pas forcément en terme économique (Laprie, Deswartes, et al. 1996).

Nous pouvons alors distinguer trois types de systèmes critiques (Sommerville 2011) en fonction du type de coût que leur défaillance entrainerait :

- Les systèmes critiques *par rapport à la vie humaine* (`safety-critical`), pour qui une défaillance risque d'affecter des vies humaines ou de créer un dommage environnemental important. Nous pouvons par exemple citer les systèmes de transport commerciaux comme les avions civils dont la défaillance peut entrainer le crash de l'avion et la perte de nombreuses vies.
- Les systèmes critiques *par rapport à la mission* (`mission-critical`), pour qui une défaillance pourrait résulter en la défaillance de la mission. Nous pouvons par exemple citer un robot ayant une mission dans un endroit inaccessible pour l'homme dont la défaillance peut entrainer l'échec de la mission.
- Les systèmes critiques *par rapport à l'activité entrepreneuriale* (`business-critical`), pour qui une défaillance pourrait résulter en une défaillance de l'entreprise qui l'utilise. Nous pouvons par exemple citer un système grand public tel qu'une voiture dont une défaillance mineure (ne pouvant pas provoquer la mort des passagers, telles qu'un problème dans le bloc de démarrage lorsque l'on gare la voiture sous ligne haute tension) peut nuire à la notoriété de l'entreprise la commercialisant.

2.1.3 Systèmes interactifs critiques

La notion de *système interactif critique* a été proposée par (Palanque et Bastide 1994) et correspond à un système interactif pour lequel une défaillance a un coût supérieur à celui du développement du système. De la même manière que les systèmes critiques, les systèmes interactifs critiques peuvent être classifiés en trois catégories :

- *Les systèmes interactifs critiques par rapport à la vie humaine.* Nous pouvons par exemple citer les systèmes de commande et contrôle d'un sous-marin dont la défaillance peut entrainer l'impossibilité de remonter à la surface et donc la mort des passagers.
- *Les systèmes interactifs critiques par rapport à la mission*. Nous pouvons par exemple citer les systèmes de commande et contrôle d'une sonde spatiale dont la défaillance peut provoquer une sortie de la sonde de son orbite.
- *Les systèmes interactifs critiques par rapport à l'activité entrepreneuriale*. Nous pouvons par exemple citer une télécommande ou un téléphone portable dont la défaillance peut nuire à la notoriété de l'entreprise la commercialisant.

Ainsi, les systèmes interactifs critiques ont une portée très large et couvrent de nombreux domaines d'application : ils peuvent aller des jeux vidéo aux systèmes de commande et contrôle des systèmes spatiaux tout en passant par les télévisions interactives et les postes de conduite des voitures et des avions. Tous ces domaines sont très disparates mais ils concernent tous des systèmes interactifs critiques pour lesquels la fiabilité du système est nécessaire. En effet, pour tous ces systèmes, une défaillance serait très coûteuse : par exemple, si un jeu vidéo ne fonctionne pas, l'utilisateur ne s'amusera pas avec et le jeu ne se vendra pas aussi bien que prévu et il en résultera une perte conséquente pour le fabriquant. La criticité du système est également très dépendante du point de vue de l'utilisateur et du contexte d'utilisation : par exemple, un utilisateur achetant un téléphone tactile pour le travail considèrera beaucoup plus ennuyeuse une défaillance de celui-ci qu'un utilisateur possédant plusieurs téléphones.

2.2 Propriétés des systèmes interactifs et des systèmes critiques

2.2.1 Propriétés des systèmes interactifs

La conception des systèmes interactifs fait l'objet d'exigences et de besoins particuliers par rapport à la conception de systèmes informatiques non-interactifs. Ces propriétés font l'objet de plusieurs travaux, standards et recommandations. Nous présentons dans les sous-sections suivantes les principales propriétés des systèmes interactifs : l'ensemble des *propriétés permettant de garantir la qualité du système*, *l'utilisabilité* du système et enfin le *ressenti utilisateur*, plus communément connu sous sa dénomination anglaise : *User eXperience* ou *UX*.

2.2.1.1 Propriétés permettant de garantir la qualité des systèmes interactifs

Les travaux de (Gram et Cockton 1996) ont identifié les différentes propriétés à respecter afin de garantir la qualité des systèmes interactifs. La sélection et la prise en compte des propriétés nécessaires au système interactif doivent être faites durant la conception et le développement du système. La Figure 2.3, extraite de (Martinie De Almeida 2011), représente ces différentes propriétés, classifiées en deux catégories : les propriétés externes liées à la qualité du système d'un point de vue de l'utilisateur et les propriétés internes liées à la qualité du système du point de vue de sa conception, de son ingénierie.

Les propriétés externes

Elles qui sont liées à l'utilisabilité du système et permettent de caractériser la qualité de l'interaction entre l'homme et la machine, en garantissant que le système est agréable à utiliser, fiable, compréhensible et permet d'accomplir les tâches nécessaires à l'utilisateur. Elles se divisent en deux catégories qui sont la flexibilité de l'interaction qui permet de qualifier la manière dont le système échange des informations avec l'utilisateur et la robustesse de l'interaction, qui permet de qualifier la capacité du système à permettre à l'utilisateur d'accomplir les tâches qu'il a à faire sans commettre

d'erreurs irréversibles. Ces propriétés sont nombreuses car elles sont très fortement liées à l'utilisateur et à son imprédictibilité qui rend son comportement et ses actions imprévisibles ; elles relèvent surtout du design du système interactif.

Les propriétés internes

Elles sont liées à la conception (autant logicielle que matérielle) du système. Elles sont très fortement liées aux propriétés classiques de l'ingénierie de conception. Elles permettent par exemple de garantir la capacité d'évolution du système, sa maintenabilité, ses performances et son exhaustivité fonctionnelle (sa capacité à fournir à l'utilisateur les fonctions dont il a besoin). Ces propriétés relèvent du développement du système et leur garantie peut-être assurée par l'utilisation d'architectures logicielles spécifiques aux systèmes interactifs telles que celles que nous présentons en section 2.4.

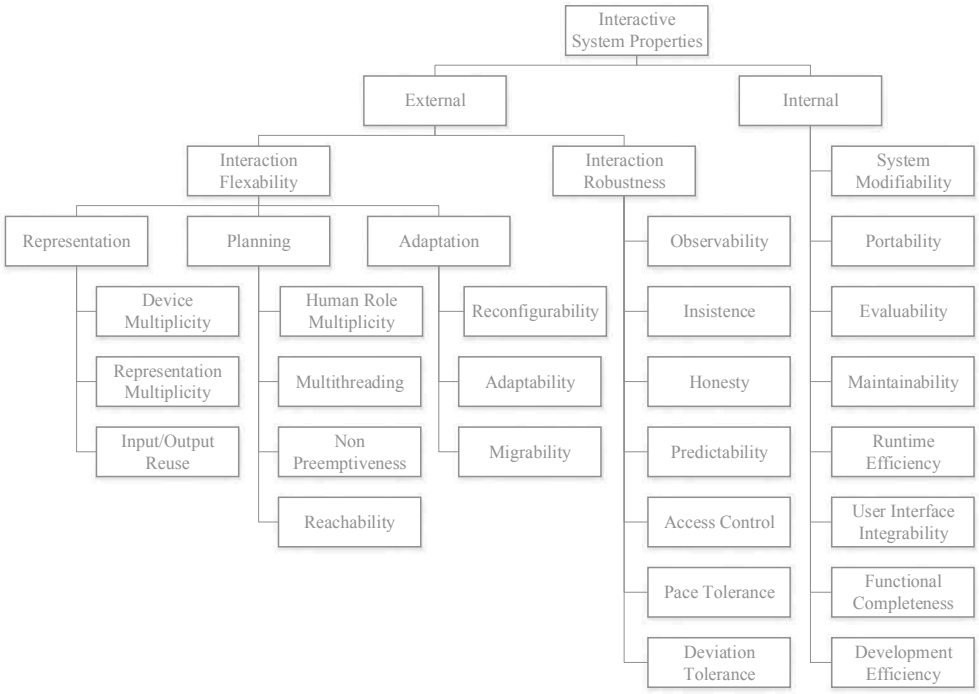


Figure 2.3. Propriétés externes et internes d'un système interactif

2.2.1.2 L'utilisabilité

L'utilisabilité d'un système interactif est définie par la norme ISO 9241 (International Standard Organization 1996) comme « *le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation spécifié* ». Cette propriété se décompose donc en trois facteurs :

- *L'efficacité* est la capacité à atteindre le but prévu ;
- *L'efficience* est la capacité à atteindre le résultat prévu avec un effort et un temps minimal ;
- *La satisfaction* correspond au confort et au ressenti de l'utilisateur lors de son interaction avec le système.

L'utilisabilité est une propriété fondamentale pour les systèmes interactifs. En effet, si un système n'est pas utilisable, l'utilisateur ne pourra pas s'en servir, et donc, par extension, le système sera incapable de remplir le service désiré et ne fonctionnera pas, comme le suggère Susan Dray dans son slogan « *If the user can't use it, it doesn't work* ».

Cette propriété est aujourd'hui largement répandue et mise en œuvre dans la communauté de recherche sur les interfaces homme-machine (IHM) qui a développé de nombreuses méthodes pour évaluer l'utilisabilité d'un système telles que la réalisation de tests avec des utilisateurs, des évaluations heuristiques ou encore des méthodes basées sur les modèles (Nielsen et Mack 1994).

Il est important de ne pas considérer l'utilisabilité d'un système seulement lorsque celui-ci est développé et de la prendre en compte tout au long du développement. Cela peut être réalisée en prenant en compte l'analyse des besoins utilisateurs lors de la conception du système et en utilisant des processus itératifs permettant de prendre compte l'utilisabilité du système lors de différentes phases du développement.

L'utilisabilité d'un système est évaluée grâce à l'évaluation des trois propriétés qui la constitue :

- *L'efficacité* peut être évaluée grâce à une analyse des tâches utilisateurs. En effet, celle-ci permet de déterminer tous les services devant être proposés par le système. Une mise en correspondance entre les tâches utilisateurs et les fonctionnalités proposées par le système permet donc de vérifier l'efficacité de celui-ci.
- *L'efficience* peut être évaluée grâce à des tests utilisateurs en mesurant des variables physiques telles que le temps requis pour effectuer une tâche. Elle peut également être évaluée en estimant la charge de travail de l'utilisateur en utilisant par exemple l'index NASA TLX (pour NASA Task Load Index) (Hart et Staveland 1988).
- *La satisfaction* est le plus souvent évaluée à l'aide de test utilisateurs et de questionnaires tels que le questionnaire SUS (pour System Usability Scale) (Brooke 1996).

Concernant les tests utilisateurs, les travaux de Nielsen (Nielsen et Mack 1994) ont montré qu'une étude avec cinq utilisateurs pouvait détecter 80% des problèmes d'utilisabilité (voir Figure 2.4). C'est pour cela que Nielsen promeut le concept de « *discount usability engineering* » (Nielsen 2009) en soulignant qu'une étude d'utilisabilité impliquant des tests avec cinq utilisateurs permet d'assurer, à moindre coût, que le système est suffisamment utilisable.

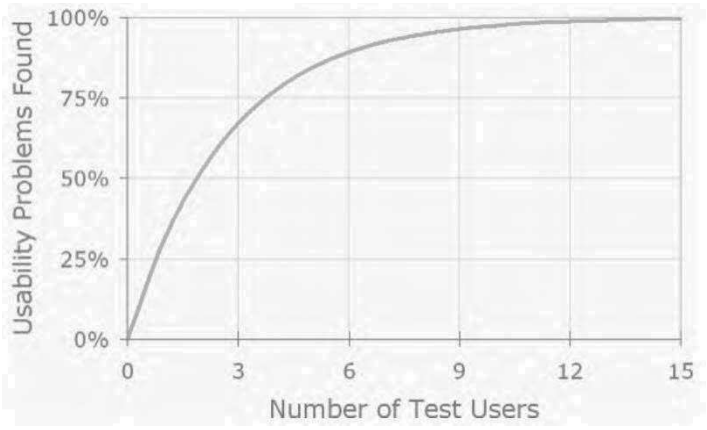


Figure 2.4. Problèmes d'utilisabilités détectés en fonction du nombre de test utilisateurs (Nielsen 2000)

2.2.1.3 L'User eXperience (UX)

L'*User eXperience* est l'une des propriétés des systèmes interactifs les plus étudiées à l'heure actuelle par la communauté scientifique des interfaces homme-machine. Elle est définie par la norme ISO 9241 (International Standard Organization 1996) comme *« les perceptions et les réactions d'une personne qui résultent de l'utilisation effective et/ou anticipée d'un produit, système ou service ».* Cependant, comme le souligne les travaux de (Bernhaupt 2015), cette définition se concentre sur la perception de l'utilisateur et ne permet pas de donner des moyens d'évaluer l'user experience. Il est important pour cela de clarifier les différents dimensions et facteurs de cette propriété afin de permettre son évaluation. Les travaux de (Bernhaupt 2015) résume les dimensions de l'user experience qui ont été traitées par les communautés de recherche de l'interaction homme-machine et des jeux. Cette propriété peut ainsi se diviser en 4 dimensions fondamentales :

- *Esthétique* : décrit combien le système est perçu esthétiquement agréable ou beau.
- *Émotion* : décrit combien le système peut provoquer d'émotions, peut affecter l'utilisateur.
- *Stimulation* : décrit combien le système peut donner à l'utilisateur des services innovateurs et intéressants.

- *Identification* : décrit combien le système peut permettre à l'utilisateur de s'y identifier.

Ces dimensions fondamentales sont associées avec d'autres facteurs tels que la valeur et le sens, le lien social, la sûreté de fonctionnement, la sécurité et la confiance, la qualité de service, l'immersion, l'implication, l'engagement ou encore la jouabilité.

Comme le mettent en évidence les travaux de (Bernhaupt 2015), l'user experience doit être, de la même manière que l'utilisabilité, prise en compte tout au long du processus de développement du système, depuis sa spécification jusqu'à sa mise en service.

2.2.2 Propriétés des systèmes critiques

Dans le domaine des systèmes critiques, (Laprie, Deswartes, et al. 1996) et (Avizienis, Laprie, et al. 2004) définissent les concepts de sûreté de fonctionnement (`dependability`) et sécurité (`security`). Ces deux propriétés sont essentielles pour les systèmes critiques.

La *sûreté de fonctionnement* d'un système est définie comme la propriété de celui-ci permettant à ses utilisateurs (pouvant être des humains ou d'autres systèmes) de placer une confiance justifiée dans le service qu'il leur délivre.

La *sécurité* d'un système est définie comme la propriété de celui-ci à se protéger des attaques malveillantes, visant à lui nuire (autant par la création de défaillances du système que par le vol d'informations confidentielles).

Ces deux concepts sont très liés et ont été raffinés dans un arbre présentée en Figure 2.5 permettant de mettre en évidence les trois grands axes qui les constituent :

- *Les attributs (`attributes`)* : les propriétés qui les définissent.
- *Les entraves (`threats`)* : les circonstances indésirables, leurs sources et leurs résultats, provoquant la non-sûreté de fonctionnement ou non-sécurité.
- *Les moyens (`means`)* : les méthodes et techniques permettant de garantir au mieux la sûreté de fonctionnement et la sécurité.

Afin d'avoir un meilleur aperçu de ces axes, nous les détaillons dans les sous-sections.

Les attributs

Comme présenté en Figure 2.5 dans le premier sous arbre, la sûreté de fonctionnement et la sécurité sont composées de diverses propriétés complémentaires. Ces propriétés permettent, en fonction du système considéré, de mettre l'accent sur les aspects les plus pertinents de ces deux notions en fonction des applications auxquelles le système est destiné. On distingue ainsi :

- *La disponibilité (`Availability`)* : capacité du système à être prêt pour son utilisation.
- *La fiabilité (`Reliability`)* : capacité du système à fournir un service correct et continu.
- *La sécurité-innocuité (`Safety`)* : capacité du système à éviter les conséquences catastrophiques pour son environnement et ses utilisateurs.
- *La confidentialité (`Confidentiality`)* : capacité du système à garantir la non divulgation d'informations confidentielles (cette propriété concerne seulement la sécurité).
- *L'intégrité (`Integrity`)* : capacité du système à garantir l'absence d'altération de son information et de son fonctionnement (cette propriété concerne seulement la sécurité).
- *La maintenabilité (`Maintainability`)* : capacité du système à être modifiable de manière à permettre sa réparation et ses évolutions.

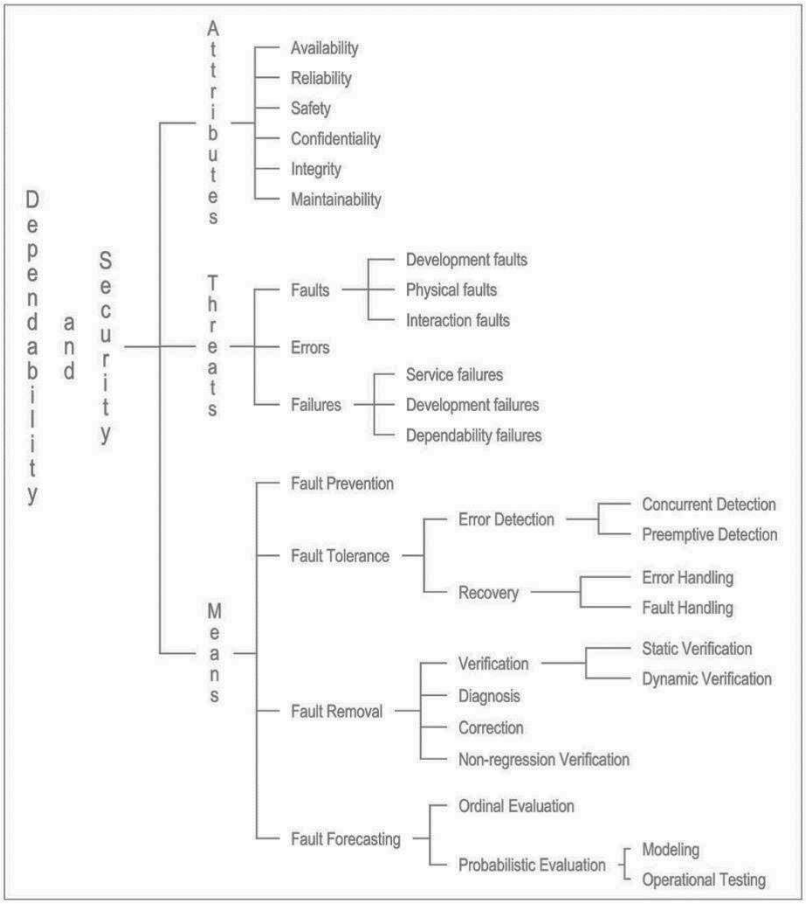


Figure 2.5. Arbre des concepts de sûreté de fonctionnement et de sécurité (Avizienis, Laprie, et al. 2004).

Les entraves

Le respect des propriétés précédentes permet de concevoir des systèmes sûrs de fonctionnement. Cependant, ces propriétés peuvent être corrompues, menant ainsi à des circonstances indésirables provoquant la non-sûreté de fonctionnement. Ces circonstances ainsi que leurs sources et leurs effets, sont définis comme les entraves à la sûreté de fonctionnement et à la sécurité. On distingue trois types d'entraves : les fautes (`Faults`), les erreurs (`Errors`) et les défaillances (`Failures`).

La *défaillance* d'un système correspond à l'occurrence d'un comportement inacceptable du système par rapport à son comportement attendu. Une *erreur* est une partie de l'état du système susceptible de causer une défaillance. Les *fautes* quand à elles, sont les causes avérées ou hypothétiques des erreurs. Comme le montre la Figure 2.6, ces trois entraves à la sûreté de fonctionnement sont liées par des effets de causalité. Ainsi, une faute est dormante tant qu'elle ne provoque pas d'erreur ; lors de son activation, elle déclenche une erreur. Par propagation, une erreur peut mener directement à une défaillance (affectant alors le service du système) ou à plusieurs erreurs provoquant par la suite une défaillance. Cependant, une erreur peut également disparaître avant de provoquer la moindre défaillance. On remarque également sur la Figure 2.6 qu'une défaillance peut provoquer une faute. En effet, tout dépend de la frontière donnée au système : une défaillance affectant un composant A implique que le service qu'il fournit dévie de sa fonctionnalité. Ainsi, un composant B, utilisant le service du composant A, observera la défaillance du composant A comme une faute extérieure pouvant mener à une erreur puis à une défaillance et ainsi de suite.

Figure 2.6. Chaîne de causalité des entraves à la sûreté de fonctionnement et à la sécurité (Avizienis, Laprie, et al. 2004).

Les fautes pouvant affecter les systèmes informatiques peuvent être extrêmement diverses. Les travaux de (Avizienis, Laprie, et al. 2004) en proposent une classification exhaustive afin de permettre d'identifier toutes les classes de fautes pouvant affecter un même système. Cette classification est faite sous forme d'arbre et met en évidence les différents critères permettant d'identifier 31 classes de fautes élémentaires. Nous retiendrons ainsi : leur phase de création ou d'occurrence (durant le développement ou durant la phase opérationnelle), leur situation par rapport aux frontières du système (interne ou externe), leur cause phénoménologique (dues à l'homme ou dues à des phénomènes naturels), leur dimension (affectant le matériel ou le logiciel), leur objectif (malveillant ou non malveillant), leur persistance (fautes permanentes ou temporaires).

Les moyens

Pour limiter l'apparition d'erreurs et de défaillances, il s'agit de traiter leur source : les fautes. Dans ce but, plusieurs méthodes et techniques ont été identifiées et (Avizienis, Laprie, et al. 2004) propose de les classifier en quatre catégories (voir Figure 2.5) :

- *La prévention des fautes (`fault prevention`)* : permet d'éviter autant que possible l'introduction de fautes pendant le développement du système. On emploie généralement des techniques de développement rigoureuses, de la formalisation, de la modélisation, …
- *La tolérance aux fautes (`fault tolerance`)* : permet d'éviter la défaillance du système en présence de fautes via la détection des erreurs qu'elles causent et leur recouvrement. La *détection* permet d'identifier la présence d'erreur, leur type et leur cause. Le *recouvrement* d'erreur vise à transformer l'état du système contenant une ou plusieurs erreurs en un état sans erreur de manière à ce que le service fourni par le système puisse toujours être assuré.
- *L'élimination de fautes (`fault removal`)* : permet de réduire le nombre de fautes pouvant survenir à la fois pendant le développement du système (généralement en utilisant des techniques de vérification de propriétés, de preuve, model-checking, de test, …) et pendant la phase d'exploitation du système (par exemple en effectuant de la maintenance corrective).
- *La prévision des fautes (`fault forecasting`)* : consiste à estimer le nombre, l'incidence et les conséquences probables de fautes, généralement en dressant une évaluation statistique de la fréquence et de l'impact des fautes.

Mesure de la sûreté de fonctionnement

La sûreté de fonctionnement peut être évaluée au travers l'évaluation de ces différents attributs et plus particulièrement de ces attributs de fiabilité, disponibilité et maintenabilité (Laprie, Deswartes, et al. 1996). Ces mesures sont généralement faites à l'aide de définitions probabilistiques et d'estimateurs statistiques tels que le temps moyen jusqu'à la première défaillance, la fiabilité initiale ou le taux de défaillance initial.

Le choix des critères sur lesquels porter un intérêt particulier pour la mesure de sûreté de fonctionnement se fait en fonction du domaine d'application des systèmes. Ainsi pour un réseau téléphonique on privilégiera prioritairement la disponibilité, pour une sonde spatiale la fiabilité et pour une centrale nucléaire la sécurité-innocuité.

Pour les systèmes avioniques embarqués, le critère privilégié est la sécurité-innocuité. Il s'agit alors dans ce cas de garantir une probabilité d'occurrence de défaillance catastrophique par heure de vol inférieure à 10^{-9}/h pour tous les systèmes critiques.

2.2.3 Conflits de propriétés pour les systèmes interactifs critiques

Les systèmes interactifs critiques relèvent à la fois des propriétés des systèmes interactifs et de celles des systèmes critiques. Parmi toutes celles que nous avons exposées précédemment, certaines peuvent être plus importantes que d'autres en fonction du domaine d'application des systèmes concernés. Ainsi, dans le domaine des systèmes interactifs critiques par rapport à l'activité entrepreneuriale comme par exemple les télévisions interactives ou les jeux vidéos, l'user experience aura une importance primordiale alors qu'elle est reléguée à un rôle secondaire pour les systèmes

interactifs critiques par rapport à la vie humaine où l'utilisabilité est prioritaire. Dans tous les cas, la sûreté de fonctionnement reste primordiale.

Nous nous intéressons ici aux systèmes interactifs présents dans les cockpits d'avions civils et par extension aux systèmes interactifs critiques par rapport à la vie humaine, nous retiendrons donc les propriétés de sûreté de fonctionnement et d'utilisabilité comme étant les plus importantes à garantir.

Ces deux propriétés sont orthogonales et peuvent se retrouver en conflit. Ceci a été montré dans les travaux de (Martinie, Palanque, et al. 2010). Ces travaux proposent une notation pour aider les concepteurs de systèmes interactifs critiques dans leur choix de conception et mettent en évidence que certains choix de conceptions permettant de garantir la sûreté de fonctionnement du système peuvent affecter l'utilisabilité du système. De la même manière, rendre un système plus utilisable, en utilisant par exemple des techniques d'interaction avancées, peut affecter sa sûreté de fonctionnement en introduisant des fonctionnalités supplémentaires et donc en augmentant la complexité du logiciel.

Ces deux propriétés sont généralement traitées de manière séparée lors de la conception et du développement du système sans étudier leur impact mutuel. Il est fondamental de prendre en compte ces deux propriétés en les traitant de manière intégrée et systématique afin de pouvoir concevoir et développer des systèmes interactifs critiques sûrs de fonctionnement et utilisables.

Certains travaux proposent des solutions pour traiter ces deux propriétés ensemble et de manière systématique, on retrouve ainsi par exemple le processus de développement présenté dans les travaux de (Martinie, Palanque et Navarre, et al. 2012) ainsi que les travaux de (A. Tankeu-Choitat 2011) présentant une approche pour le développement de systèmes interactifs intégrant à la fois les aspects sûreté de fonctionnement et utilisabilité.

2.3 Systèmes interactifs et sûreté de fonctionnement

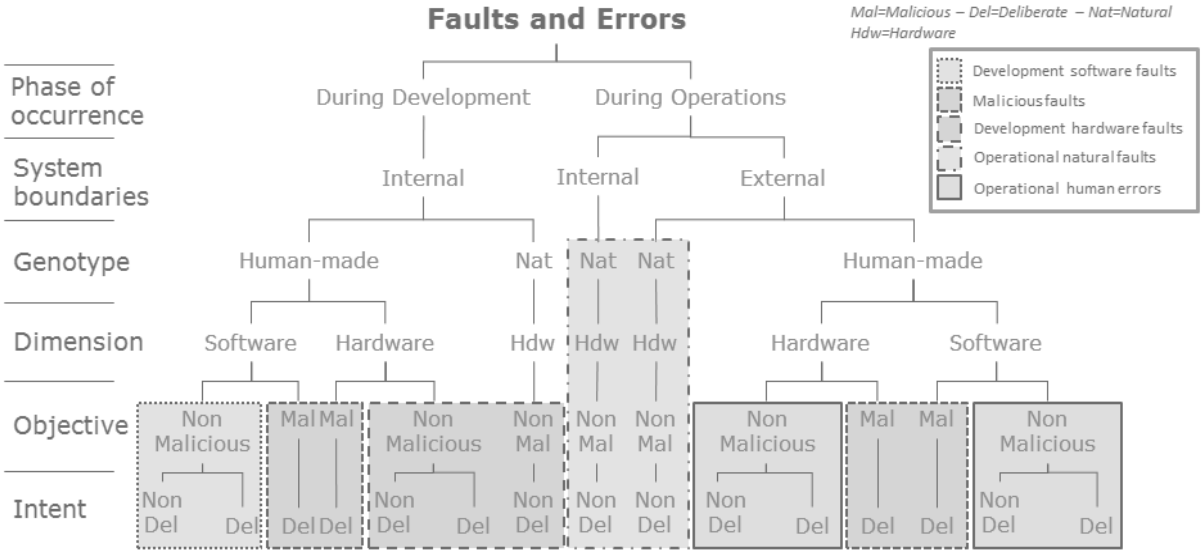


Figure 2.7. Classification des fautes des systèmes informatiques

La Figure 2.7 présente une version simplifiée de la classification des fautes pouvant affecter les systèmes informatiques proposée par les travaux de (Avizienis, Laprie, et al. 2004). Cette figure met en évidence cinq regroupements de classes de fautes que nous avons identifiés, ces regroupements peuvent être considérés comme des classes de fautes de haut niveau :

- *Fautes logicielles de développement* : fautes introduites involontairement durant le développement du système. Par exemple, les erreurs de codage ou les erreurs de conception.
- *Fautes malveillantes* : fautes introduites délibérément pour provoquer la défaillance du système. Elles relèvent de la sécurité du système. Par exemple, la prise de contrôle extérieur ou un déni de service inopiné, un crash du système.

- *Fautes matérielles de développement* : fautes ayant une cause naturelle ou humaine et impactant le matériel durant sa conception. Par exemple, un court-circuit à l'intérieur d'un processeur ou un matériel affecté lors de son développement par l'utilisation d'une eau avec une concentration trop forte en uranium, provoquant ainsi des erreurs lors de l'utilisation de celui-ci comme ce fut le cas en 1978 pour une usine d'IBM (Ziegler, et al. 1996).
- *Fautes naturelles en opération* : fautes causées par un phénomène naturel. Elles affectent le matériel (et par conséquent le logiciel) et surviennent pendant le fonctionnement du système. Par exemple, la modification de la mémoire suite à un rayonnement électromagnétique ou l'effacement d'un disque dur dans une zone magnétisée.
- *Erreurs humaines en opération* : fautes qui résultent d'une action humaine pendant le fonctionnement du système. Elles peuvent être matérielles et logicielles et peuvent être délibérées ou non. Par exemple, l'oubli d'une étape dans une procédure ou un appui sur un mauvais bouton.

Nous présentons dans les sous-sections suivantes l'état de l'existant des méthodes, outils et techniques permettant de garantir la sûreté de fonctionnement des systèmes interactifs par rapport à chacune de ces classes. Cette section n'a pas pour but de faire une présentation exhaustive de toutes les approches existantes mais de présenter une vue d'ensemble de l'état de l'art des approches pour la sûreté de fonctionnement des systèmes interactifs par rapport à ces cinq classes de fautes.

Nous nous intéressons dans cette thèse plus particulièrement aux fautes logicielles de développement et aux fautes naturelles en opération (voir Chapitre 4). C'est pour cela que nous détaillons en section 2.5 les différentes notations formelles pour la description des systèmes interactifs permettant de prévenir les fautes logicielles de développement et en section 2.6 les différentes techniques de tolérance aux fautes utilisées dans les systèmes critiques pour traiter les fautes naturelles en opération.

2.3.1 Fautes logicielles de développement

Les fautes logicielles de développement sont généralement prises en compte et traitées en utilisant des approches de prévention des fautes, élimination des fautes et tolérance aux fautes.

La communauté de l'ingénierie des systèmes interactifs à réalisé de nombreux travaux de recherche pour raffiner et étendre les approches classiques d'ingénierie du logiciel à l'ingénierie des systèmes interactifs. En effet, celles-ci ne peuvent pas être réutilisées telles quelles pour les systèmes interactifs du fait des spécificités de ces systèmes (Wasserman 1981) et (Wegner 1997) et des propriétés qu'ils doivent respecter (voir section 2.2.1).

Les approches que nous présentons ici ont, pour la plupart, été créées dans un but d'amélioration de l'utilisabilité des systèmes interactifs mais peuvent cependant être reprises dans un but d'amélioration de la sûreté de fonctionnement des systèmes. Dans ce cas, elles peuvent être utilisées comme des moyens de prévention et d'élimination des fautes logicielles de développement. Nous pouvons par exemple citer :

- *L'utilisation de processus spécialisés* comme les processus de conception centrée utilisateur tels le processus User-Centered System Design (Göransson, Gulliksen et Boivie 2003) qui permet d'identifier les problèmes d'utilisabilité tout au long du processus de développement et tout au long du cycle de vie du système ; ou le processus proposé par les travaux de (Martinie, Palanque et Navarre, et al. 2012) qui intègre la prise en compte de l'utilisabilité et de la sûreté de fonctionnement.
- *L'utilisation de méthodes formelles* (A. J. Dix 1991). Elles peuvent aller par exemple de la modélisation de l'utilisateur à l'aide de notations formelles telles que GOMS (Card, Newell et Moran 1983) ou HAMSTERS (Martinie De Almeida 2011) à la modélisation du système (voir section 2.5).
- *L'utilisation de techniques de test* qui permettent de vérifier la correspondance entre le système et les spécifications ; comme par exemple les travaux de (Memon, Pollack et Soffa 2001) qui proposent de générer automatiquement des cas de test pour les interfaces graphiques.
- *L'utilisation de standardisations* comme par exemple le standard CUA créé par IBM (IBM 1989) qui standardise les interfaces WIMP.

La tolérance aux fautes logicielles n'a que très peu été abordée dans le domaine des systèmes interactifs critiques et seuls les travaux de (A. Tankeu-Choitat 2011) que nous détaillons dans la section 2.3.4, permettent, dans une moindre mesure, de tolérer certaines fautes logicielles.

2.3.2 Fautes malveillantes

Les fautes malveillantes concernent la sécurité des systèmes interactifs. Cet aspect a été très étudié dans le domaine des interactions homme-machine. Cependant, encore une fois, cet aspect a été étudié par la communauté dans un souci d'utilisabilité. Les questions d'intégrité et de protection des données sont laissées aux domaines liés à la sécurité informatique mais celles-ci n'ont pas encore été traitées spécifiquement pour les systèmes interactifs.

Ceci est illustré par la création d'un symposium sur la sécurité et la protection de la vie privée utilisables. Ce symposium, appelé SOUPS (Symposium on Usable Privacy and Security), permet de discuter les questions d'utilisabilité des systèmes d'authentification. Nous pouvons par exemple citer les travaux de (Wright, Patrick et Biddle 2012) qui proposent la reconnaissance de texte comme moyen d'authentification, facilitant ainsi le travail de mémoire de l'utilisateur.

Dans le domaine des systèmes critiques tels que les systèmes avioniques embarqués, les questions d'intégrité et de sécurité des données reposaient jusqu'à présent sur le fait que ces systèmes sont fermés et non accessibles au public. Cependant, avec l'introduction de nouvelles technologies comme les réseaux informatiques dans les avions, ces questions commencent à se poser pour les systèmes embarqués comme le montrent les travaux de (Dessiatnikoff, et al. 2013) qui mettent en évidence comment certaines attaques peuvent affecter le bon fonctionnement d'un système informatique embarqué avionique.

2.3.3 Fautes matérielles de développement

Les composants matériels utilisés pour les systèmes interactifs sont similaires à ceux utilisés dans les systèmes informatiques conventionnels. La prise en compte des fautes de développement matériel peut donc être traitée avec des processus de développement usuels, comme ceux décris dans la norme DO-254 (RTCA et EUROCAE 2000), qui fournit un guide de développement des composants électroniques embarqués.

Cependant, dans le cas d'un système interactif, il est nécessaire, en plus de prendre en compte la sûreté de fonctionnement des composants informatiques classiques comme la mémoire ou le processeur, prendre en compte la sûreté de fonctionnement des périphériques d'entrée et sortie. Par exemple, l'introduction dans un cockpit avionique d'un nouveau type de périphérique d'entrée-sortie tel qu'un écran tactile nécessite que ce matériel ait un niveau de sûreté de fonctionnement suffisant : c'est-à-dire une probabilité de défaillance par heure de vol de moins de 10^{-9}/h.

Le concepteur du système interactif pourra néanmoins augmenter la fiabilité d'un système interactif par rapport à l'introduction d'un nouveau matériel en adaptant le logiciel sous-jacent. Ainsi, si l'on reprend l'exemple de l'introduction d'un écran tactile dans un cockpit d'avion, la sûreté de fonctionnement de ce matériel peut être augmentée en imposant des techniques d'interactions particulières : une interaction oblique dépend de plusieurs lignes de la grille tactile, ce qui la rend sûre par rapport à la défaillance d'une de ces lignes contrairement à une interaction linéaire horizontale.

2.3.4 Fautes naturelles

La prise en compte des fautes naturelles nécessite la mise en place de mécanismes spécifiques pour les tolérer car elles ne sont ni prévisibles ni évitables. Peu de contributions concernant la prise en compte de ces fautes sont disponibles dans le domaine des systèmes interactifs. On peut cependant trouver les travaux de (D. Navarre, P. Palanque, et al. 2008) qui présentent la reconfiguration dynamique de la technique d'interaction ou la possibilité de la réorganisation des écrans quand les défaillances matérielles le nécessitent.

Cependant, cette solution n'est pas suffisante et ces aspects ne peuvent être ignorés lorsque l'on traite de la fiabilité de systèmes interactifs critiques. Ceci est particulièrement vérifié pour les systèmes

interactifs critiques tels que ceux déployés en haute atmosphère (avions) ou dans l'espace (vol spatiaux habités) car une plus forte probabilité d'occurrence de fautes concerne ces systèmes (Normand 1996b). Ces fautes naturelles démontrent la nécessité d'aller plus loin que la prévention classique des fautes durant le développement en prenant en compte les fautes pouvant apparaitre en opération.

Dans cet objectif, les travaux de (Tankeu-Choitat, et al. 2011) et (A. Tankeu-Choitat 2011) proposent d'intégrer des mécanismes de tolérance aux fautes aux composants de base des systèmes interactifs : les widgets. Pour cela, ces travaux proposent d'appliquer une architecture autotestable aux widgets (voir les différentes architectures de tolérance aux fautes en section 2.6). Cette architecture s'appuie sur la duplication des fonctionnalités de chaque widget. Un widget est donc alors composé de deux composants logiciels effectuant en parallèle les mêmes fonctionnalités à partir des mêmes entrées. Leurs résultats sont analysés et comparés par un comparateur. Celui-ci permet de détecter une erreur si les résultats des deux composants sont différents.

2.3.5 Erreurs humaines

Les erreurs humaines sont étudiées depuis de nombreuses années et ont été classifiées par les travaux de (Reason 1990). Les erreurs humaines peuvent être prévenues ou tolérées (Dearden et Harrison 1995) et (Laprie, Deswartes, et al. 1996). De nombreux moyens ont été développés pour la prise en compte de cette problématique. Nous citerons par exemple :

- *Le développement de manuel d'utilisation* comme présenté par les travaux de (Bowen et Reeves 2012).
- *La formation des utilisateurs* à l'aide de formations spécialisées qui peuvent être créées et supportées au moment du développement du système comme proposé par les travaux de (Martinie, Palanque et Navarre, et al. 2011).
- *La création d'une aide contextuelle* (Palanque, Bastide et Dourte 1993).
- *La proposition d'un meilleur design* du système interactif comme par exemple les travaux de (Thimbleby et Gimblett 2011) qui présentent une approche pour une meilleure implémentation des entrées de données numériques afin d'éviter les erreurs de saisie.

2.4 Les architectures logicielles pour les systèmes interactifs

Il existe plusieurs modèles d'architecture logicielle spécifiques aux systèmes interactifs. Ces modèles peuvent être regroupés en deux catégories : les modèles linguistiques ou à couches décrivant la structure globale des applications sous forme de couches logiques et les modèles à agent proposant une décomposition modulaire de l'interface en un ensemble d'agents communiquants (Dragicevic 2004). Nous décrivons dans les sous-sections suivantes ces deux groupes de modèles ainsi que leurs principaux représentants.

2.4.1 Les modèles linguistiques

Les modèles linguistiques découlent d'une approche linguistique de l'interaction. Ils définissent les couches logiques de l'interaction en identifiant ses trois composantes :

- *Composante lexicale* : correspond à un vocabulaire d'entrée (par exemple, le clic) et de sortie (par exemple, une icône).
- *Composante syntaxique* : correspond à une syntaxe représentant les séquences d'actions valides.
- *Composante sémantique* : correspond à la partie fonctionnelle et non-interactive du système.

Seeheim

Le modèle d'architecture Seeheim (Pfaff 1985) est le premier modèle d'architecture à structurer les applications interactives en trois composants logiques correspondants aux trois composantes décrites ci-dessus. Figure 2.8 présente ce modèle et permet de mettre en évidence ces trois composants :

- *La présentation* : cette partie correspond à la composante lexicale. Elle permet de gérer l'interaction avec l'utilisateur en interprétant les actions de celui-ci et en générant les sorties qu'il peut percevoir.
- *Le dialogue* : cette partie correspond à la composante syntaxique. Elle permet de gérer les échanges entre l'utilisateur et le système en maintenant une représentation graphique de l'état du système et des actions rendues possibles à l'utilisateur.
- *L'interface au noyau fonctionnel* : cette partie correspond à la composante sémantique. Elle permet de convertir les actions de l'utilisateur en appels de fonctions sur le noyau fonctionnel ainsi qu'une présentation de l'état du noyau fonctionnel à l'utilisateur.

En plus de ces trois composants, ce modèle introduit un composant supplémentaire, plus abstrait, permettant de représenter le *retour sémantique rapide* qui est rendu à l'utilisateur. Celui-ci est plus communément connu sous sa dénomination anglaise : *feedback* et représente les modifications immédiates du rendu graphique telles que la mise à jour du manipulateur de souris (du curseur graphique).

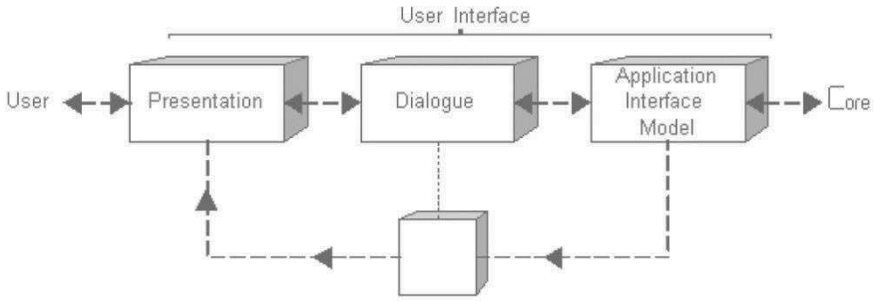


Figure 2.8. Modèle d'architecture Seeheim (Pfaff 1985)

Arch/Slinky

Le modèle architectural ARCH et son méta-modèle Slinky (Bass, et al. 1992) ont été développés en 1992 par un groupe de travail désireux de combler les lacunes des modèles architecturaux existants pour les systèmes interactifs. Ils étendent le modèle architectural Slinky.

La Figure 2.9 représente le modèle architectural ARCH qui décompose les systèmes interactifs en cinq composants que nous retrouvons sur la figure de gauche à droite :

- *Le noyau fonctionnel* encapsule les fonctions non interactives de l'application. Il contrôle et manipule les objets et données du domaine de l'application sans se soucier de la manière dont l'information sera rendue à l'utilisateur.
- *L'adaptateur du noyau fonctionnel* traduit les données provenant du noyau fonctionnel en données compréhensibles par le contrôleur de dialogue et il traduit également les données provenant du contrôleur de dialogue en données compréhensibles par le noyau fonctionnel.
- *Le contrôleur de dialogue* assure le séquencement des tâches : il décrit, en fonction de l'état du système, l'ensemble des tâches autorisées ainsi que l'effet de l'exécution de celles-ci.
- *Le composant d'interaction logique* (ou *présentation*) traduit les informations fournies par le composant d'interaction physique en informations indépendantes de l'interface (indépendantes du type d'objets physiques utilisés) et les transmet au contrôleur de dialogue. De la même manière, il traduit les informations du contrôleur de dialogue en informations spécifiques aux objets physiques utilisés.
- *Le composant d'interaction physique* (ou *boîte à outils*, *toolkit*) permet de gérer l'interaction au plus bas niveau (le niveau lexical). Il transmet les entrées de l'utilisateur sur les objets de l'interaction (par exemple les widgets) au composant d'interaction logique et transforme les données provenant du composant d'interaction logique (par exemple des modifications de l'état d'un widget) en informations graphiques visualisables et perceptibles par l'utilisateur.

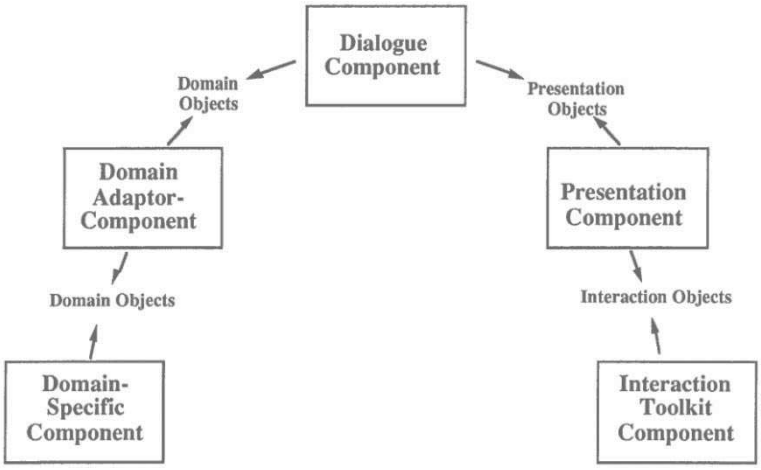


Figure 2.9. Le modèle architectural ARCH (Bass, et al. 1992)

Le modèle architectural ARCH introduit également trois types d'objets qui permettent de décrire la nature des données qui transitent entre chaque composant :

- *Les objets du domaine* décrivent les données provenant directement ou indirectement du noyau fonctionnel.
- *Les objets de présentation* décrivent de manière abstraite les événements provoqués par l'utilisateur sur les composants physiques de l'interaction ainsi que les données qui sont présentées à l'utilisateur.
- *Les objets d'interaction* sont des instances propres à un composant d'interaction physique et ils implémentent des techniques d'interaction et de visualisation qui leur sont spécifiques.

Le méta-modèle Slinky est présenté en Figure 2.10. Il a été conçu au dessus du modèle architectural ARCH afin de représenter le poids des différents composants et comment ceux-ci peuvent varier en fonction des choix fixés lors de la spécification du système interactif et des efforts portés sur un ou plusieurs composants par rapport aux autres.

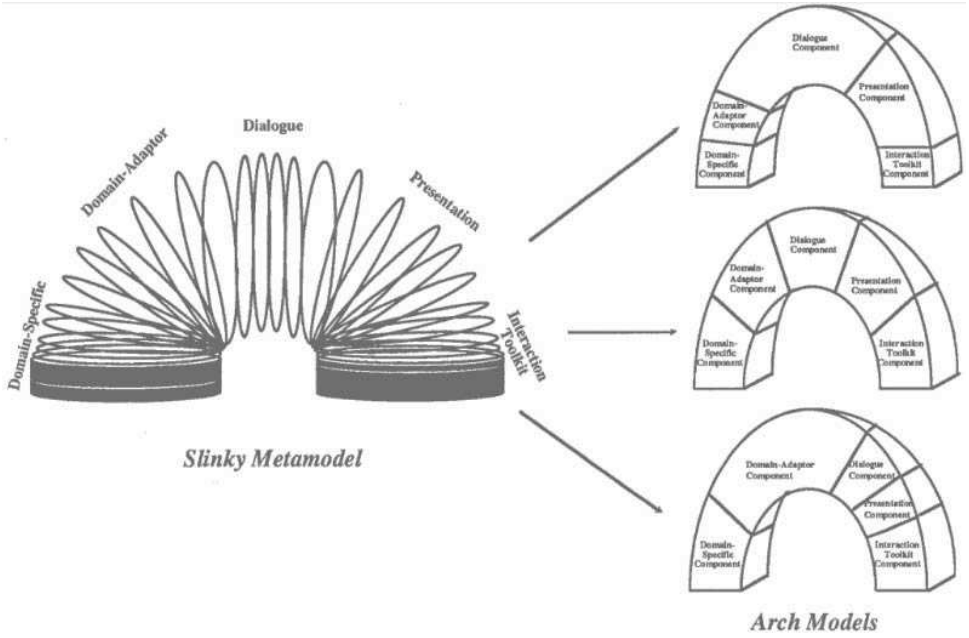


Figure 2.10. Le méta-modèle Slinky associé au modèle architectural ARCH (Bass, et al. 1992)

2.4.2 Les modèles à agents

Cette catégorie d'architecture se distingue par sa méthode de décomposition modulaire des systèmes interactifs se rapprochant des principes de la conception orientée objet. Les trois principaux

modèles architecturaux à agents sont le modèle MVC, le modèle PAC et le modèle PAC-Amodeus que nous décrivons dans les sous-sections suivantes.

MVC

Le modèle architectural MVC (pour Modèle, Vue, Contrôleur) a été défini par les travaux de (Krasner et Pope 1988) et a pour objectif de garantir un support pour une conception itérative des applications interactives. La Figure 2.11 présente ce modèle architectural et permet et mettre en évidence sa décomposition en trois composants :

- *Le modèle* : il contient les données fonctionnelles ainsi que les objets ayant un comportement complexe. Il modifie la vue associée à son état à chaque fois que celui-ci se trouve modifié par le noyau fonctionnel de l'application ou par le contrôleur.
- *Le contrôleur* : il reçoit et interprète les événements correspondants aux actions de l'utilisateur en les répercutant sur le modèle en modifiant son état ou sur la vue en effectuant le feedback.
- *La vue* : elle maintient une représentation de l'état du modèle perceptible par l'utilisateur et met celle-ci à jour lors des changements d'état du modèle.

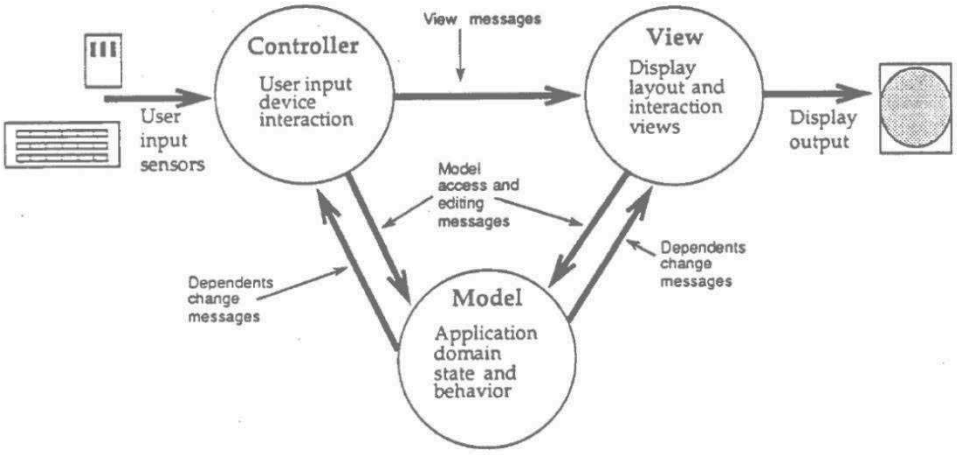


Figure 2.11. Modèle architectural MVC (Krasner et Pope 1988)

PAC et PAC-Amodeus

Le modèle architectural PAC (J. Coutaz 1987) permet de modéliser les systèmes interactifs comme une hiérarchie d'agents PAC. Cette hiérarchie, présentée en Figure 2.12 est composée de trois facettes différentes pour chaque agent PAC :

- *P pour Présentation* : cette facette décrit les entrées et sorties de l'agent perçues par l'utilisateur.
- *A pour Abstraction* : cette facette décrit les données et les méthodes de l'agent.
- *C pour Contrôleur* : cette facette assure la cohérence entre l'abstraction et la présentation et permet la communication entre les agents.

Une approche récursive permet de modéliser l'architecture complète d'une application interactive par une hiérarchie d'agents PAC. Cette hiérarchie est composée de plusieurs niveaux d'abstraction se rapprochant des couches définies par le modèle architectural Seeheim.

Le modèle architectural PAC-Amodeus (Nigay et Coutaz 1993), présenté en Figure 2.13 est un modèle hybride associant le point de vu linguistique du modèle architectural ARCH au point de vue à agents du modèle architectural PAC.

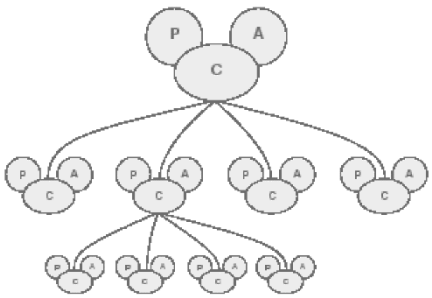


Figure 2.12. Modèle architectural PAC (J. Coutaz 1987)

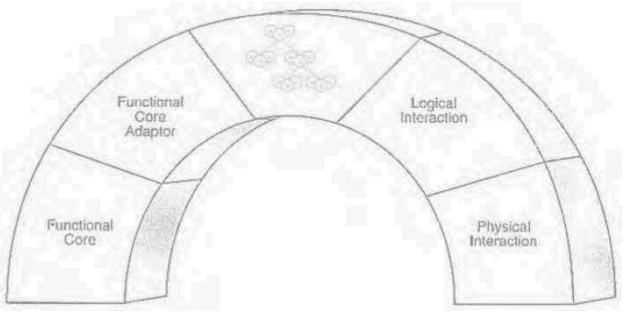


Figure 2.13. Modèle architectural PAC-Amodeus (Nigay et Coutaz 1993)

2.4.3 Une notation pour la description des architectures

Nous décrivons dans cette section le langage architectural AADL (Architecture Analysis and Design Language) (SAE International 2012) et plus particulièrement sa représentation graphique. AADL est un langage de description (textuel et graphique) standardisé en 2004 par la société SAE (Society of Automotive Engineers) pour aider à l'analyse et à la conception d'architectures système. Le langage AADL a principalement été utilisé pour décrire les systèmes temps réel complexes comme les systèmes avioniques et spatiaux mais il a aussi été utilisé dans d'autres domaines. Nous l'utilisons dans la thèse pour décrire graphiquement les architectures que nous proposons. En effet, ce langage permet de représenter à la fois les aspects matériels et logiciels, ce qui est particulièrement adaptée à la description des systèmes interactifs qui présentent une forte liaison entre ces deux aspects (voir section 2.1.1). La Figure 2.14 présente les différents composants du langage que nous utilisons dans cette thèse.

AADL component representation	Meaning of the component
System	**System:** Hierarchical organization of components.
Processor	**Processor:** Hardware component including a CPU, memory, bus ... and allowing software execution.
Virtual Processor	**Virtual processor:** Logical resource allowing software execution.
Bus	**Bus:** Provides physical connectivity between execution platform components.
Device	**Device:** Interface to external environment (typically physical components interfacing with the environment.
Event Data Ports — Output / Input	**Event Data Ports:** Ports allowing directional transfer of queued messages.
Bus access feature	**Bus access feature:** To model required access to shared bus.

Figure 2.14. Composants AADL utilisés dans la thèse

2.4.4 Synthèse

Nous venons de présenter les architectures logicielles spécifiques aux systèmes interactifs. Ces architectures permettent de garantir une partie des propriétés internes de ces systèmes telles que la modifiabilité que nous avons présentés en section 2.2.1.1. Parmi ces architectures, le modèle architectural ARCH est particulièrement intéressant car il a spécifiquement été créé pour la modifiabilité des systèmes interactifs et car il permet une décomposition fonctionnelle des systèmes interactifs qui s'avère très utile lors des étapes d'analyse de sûreté de fonctionnement.

Nous avons également présenté une notation graphique qui nous permet, tout au long de ce document, de décrire les architectures logicielles et matérielles des différents systèmes.

2.5 Les notations pour la modélisation des systèmes interactifs

Depuis les travaux précurseurs de (Parnas 1969), de nombreux travaux de recherche sont dédiés à la définition de notations pour la modélisation des systèmes interactifs. Ces notations sont couramment appelées User Interface Description Languages (UIDL) bien qu'elles ne soient pas dédiées uniquement à la description de l'interface graphique et qu'elles permettent parfois la modélisation de l'ensemble des composants du système interactif (par exemple, l'ensemble des composants décrits par le modèle architectural ARCH).

L'utilisation de ces notations permet notamment d'avoir une approche plus rigoureuse dans le développement des systèmes interactifs en levant les ambiguïtés liées au langage naturel (A. J. Dix 1995) et d'améliorer la communication entre les différents acteurs du cycle de développement (Navarre 2001).

L'ensemble de ces notations peut être classés, suivant leurs caractéristiques intrinsèques de modélisation, en quatre catégories (D. Navarre, P. Palanque, et al. 2009) :

- Les notations textuelles,
- Les notations basées sur les flux,
- Les notations basées sur les états,
- Les notations basées sur les réseaux de Petri.

Dans les sous-sections suivantes, nous présentons, pour chacune de ces catégories, les notations qui leur sont les plus représentatives afin d'en déduire leurs avantages et inconvénients. En plus de leur représentativité de leur catégorie, les notations que nous avons choisies de présenter sont particulièrement intéressantes car des études ont prouvé leur applicabilité à des études de cas conséquentes ou hors du cercle académique (par exemple des études de cas industrielles).

2.5.1 Notations textuelles

Les notations textuelles rassemblent deux catégories de notations qui sont les *algèbres de processus* et les *notations basées sur XML*, en effet, ces deux catégories ont certains points communs tels que la syntaxe imposée lors de la compilation et ont une représentation textuelle.

2.5.1.1 Les algèbres de processus

Un algèbre de processus est une notation permettant la description du comportement d'un système (autant les actions qu'il peut traiter et accomplir que les événements qu'il peut traiter et envoyer) de manière algébrique et axiomatique (Baeten 2005).

Les travaux de (Baeten 2005) et (Ladry 2010) répertorient les différentes algèbres de processus. Certaines d'entres elles ont été utilisés pour la description des systèmes interactifs : *Esterel* dans les travaux de (Fekete, Richard et Dragicevic 1998), *LOTOS* dans les travaux de (Coutaz, et al. 1993) et de (Paterno et Mezzanotte 1994), *CSP* dans les travaux de (Smith et Duke 1999) et de (Van Schooten, Donk et Zwiers 1999) et enfin *Squeak* (Cardelli et Pike 1985) qui a été spécifiquement créée pour la description de techniques d'interactions.

Exemple de Lotos

LOTOS (International Standard Organization 1988) est une notation combinant les approches des algèbres de processus et des algèbres de données. Elle a été utilisée pour décrire les systèmes interactifs tels que MATIS (Multimodal Air Traffic Information System) dans les travaux de (Coutaz, et al. 1993) et de (Paterno et Mezzanotte 1994).

Un système modélisé à l'aide de LOTOS est constitué d'un ensemble de processus communiquant entre eux grâce à des portes. Chaque processus est constitué d'un ensemble d'actions observables (ayant un impact à l'extérieur du processus) et d'actions cachées (n'ayant pas d'impact à l'extérieur du processus). Lorsqu'il effectue une action observable, le processus attend ou offre un ensemble de valeurs à une porte. Une interaction entre deux processus a lieu lorsque deux processus ou plus sont prêts à

effectuer la même action observable. Cette interaction est immédiate et permet une communication synchrone ainsi que des échanges de données.

Le comportement des processus est décrit grâce à des expressions algébriques appelées *expressions comportementales*. Les comportements complexes sont exprimés par composition d'expressions comportementales simples au travers l'utilisation d'en ensemble d'opérateurs définis par la notation LOTOS :

- *Opérateur de séquence* : P >> Q.
- *Opérateur de désactivation* : P [> Q.
- *Opérateur de synchronisation* : A||B.
- *Opérateur de choix* : B1 [] B2.

Enfin, les processus peuvent communiquer entre eux par deux moyens distincts :

- *Communication par passage de valeurs* : par exemple, si un processus `A` réalise l'action `a!3;` et un processus `B` réalise l'action `a?x:int;` alors la valeur `3` est passée au processus `B` au travers l'utilisation de la variable `x`.
- *Communication par synchronisation des signaux :* par exemple, si un processus `A` réalise l'action `a!x1;` et un processus `B` réalise l'action `a!x2;` et que les deux processus partagent une porte, alors il faut que les deux processus affectent des valeurs coïncidentes pour que l'événement correspondant ait lieu (`x1=x2`).

Un des principaux avantages de cette notation est qu'elle est associée à des environnements de conception offrant de nombreuses fonctionnalités telles que la détection automatique de la validité d'une spécification par rapport à la syntaxe et à la sémantique de la notation ; la simulation interactive du fonctionnement et la vérification automatiques de propriétés comme la vivacité et la sécurité. Cependant, cette technique de description formelle ne permet pas de représenter explicitement les états et les durées (la notion de quantification du temps) sont impossibles à représenter.

2.5.1.2 Les notations basées sur XML

XML (pour eXtensible Markup Language) est une notation à balises qui permet de structurer les données. Elle est composée de structures (appelées les balises) s'imbriquant de manière récursive avant de permettre la description d'une structure de donnée.

Les travaux de (Souchon et Vanderdonckt 2003) proposent une étude comparative des notations basées sur XML en fonction des possibilités qu'ils offrent (par exemple, la disponibilité d'un outil). Les travaux de (Meixner, Orfgen et Kümmerling 2013) proposent une autre étude comparative des notations basées sur XML en fonction de critères représentant leur capacité d'intégration dans un processus de développement (par exemple, le support pour du suivi de version). Ces travaux référencent plus d'une dizaine de notations telles qu'*UsiXML* (Limbourg, Vanderdonckt, et al. 2005) que nous présentons ci-dessous mais nous pouvons également citer des notations plus récentes et spécifiques aux techniques d'interactions post-WIMP telles que *GestureML* (GestureWorks 2014).

Exemple d'UsiXML

UsiXML (Limbourg, Vanderdonckt, et al. 2005) (pour USer Interface eXtensible Markup Language) est une notation basée sur XML permettant de décrire l'interface utilisateur dans des contextes multiples (par exemple, différentes techniques d'interaction).

Cette notation permet la spécification à plusieurs niveaux d'abstraction des différents composants interactifs tels que les widgets, les techniques d'interactions… Elle permet également de spécifier les interfaces utilisateurs indépendamment des caractéristiques physiques des systèmes sur lesquels elles sont implémentées.

La notation *UsiXML* repose sur un principe de transformation de modèles présenté en Figure 2.15. Cette figure met en évidence la transformation des modèles de haut niveau (modèle de tâches et de

domaines) en modèles de plus bas niveau jusqu'à la modélisation de l'interface concrète et au code de l'application. Cette notation permet également d'effectuer du reverse engineering, permettant de recréer des modèles de plus haut niveau.

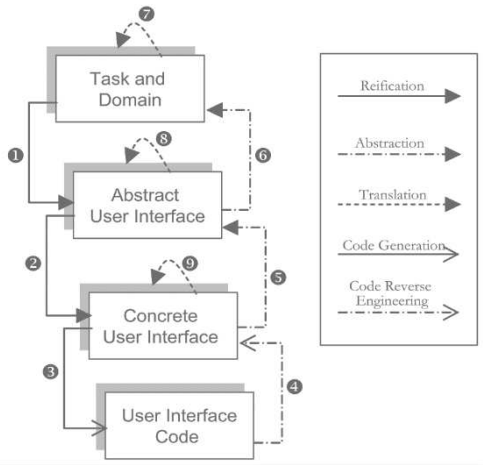


Figure 2.15. Principe de la notation UsiXML (Limbourg 2004)

2.5.2 Notations basées sur les flux

Les notations basées sur les flux permettent de décrire les évolutions des variables du système au cours de leurs traitements successifs. Ces différentes notations sont basées sur le concept de transformation d'information : ces éléments sont liés entre eux par des arcs représentants les flux. Ces flux peuvent être filtrés ou fusionnés et permettent de transmettre le résultat produit au composant suivant.

Ces notations sont souvent consacrées à la description des paramètres liés aux interactions avec l'utilisateur comme par exemple *ICon* (Dragicevic 2004) que nous présentons ci-dessous, *Squidy* (König, Rädle et Reiterer 2010) et leurs ancêtres *Marigold* (Willans et Harrison 2001) et *Whizz'Ed* (Esteban, Chatty et Palanque 1995) mais permettent également la description des widgets ou du rendu graphique.

Exemple d'ICon

ICon (pour Input Configurator) (Dragicevic 2004) est une notation associée à une boîte à outils, permettant de modéliser les entrées d'applications interactives. Elle est associée à un éditeur visuel permettant au développeur de créer rapidement des configurations d'entrées appauvries ou enrichies et pouvant être adaptées par des utilisateurs plus avancés.

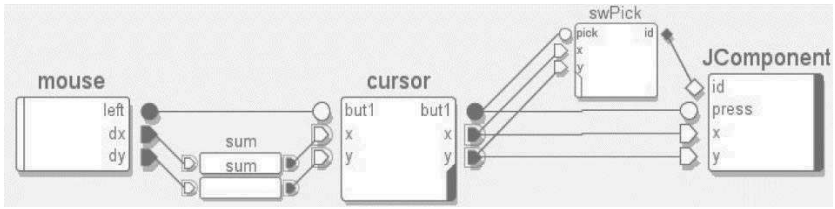


Figure 2.16. Modélisation en ICon de la sélection d'objets graphiques (Dragicevic 2004)

La Figure 2.16 présente un exemple d'utilisation de la notation ICon : la modélisation de la sélection d'objets graphiques. La notation ICon s'appuie sur une classification des dispositifs (des modules comportant des entrées et des sorties) parmi lesquels nous retiendrons en particulier :

- Les *dispositifs systèmes* représentant en général un dispositif d'entrée physique (comme le dispositif `mouse` dans l'exemple présenté en Figure 2.16).
- Les *dispositifs utilitaires* représentants tous les dispositifs autres que les dispositifs systèmes. Il peut s'agir de *dispositifs de traitement* effectuant essentiellement des transformations de données (comme

le dispositif `sum` dans l'exemple) ; des *dispositifs à retour graphiques* produisant un affichage graphique afin d'émettre des informations vers l'utilisateur (comme les dispositifs `cursor` et `JComponent` dans l'exemple).

ICon permet, grâce à son aspect visuel et sa simplicité d'utilisation (ajout de boîtes et de liens entre les ports de ces boîtes), de modéliser rapidement différentes interactions ou de modifier des modèles d'interaction existants. Cependant, il ne permet pas la représentation des états et ne peut donc pas être utilisé pour modéliser les aspects statiques du contrôleur de dialogue et de la partie applicative d'un système interactif.

2.5.3 Notations basées sur les états

Les notations basées sur les états peuvent être classées en plusieurs catégories :

- *Les automates à états finis* (ou FSM pour Finite State Machines) tels que les *Hierarchical States Machine* (Blanch et Beaudouin-Lafon 2006) que nous décrivons ci-dessous, les *SwingStates* (Appert et Beaudouin-Lafon 2006) et *HephaisTK* (Dumas, et al. 2008).
- *Les notations basées sur les états avec une notion de flux* telles que *NiMMit* (Coninx, et al. 2007) ou *IOG* (Carr 1994).
- *Les notations basées sur les états avec une notion de contraintes* telles que *ConstraintJS* (Oney, Myers et Brandt 2012).

Exemple de Hierarchical States Machine

Hierarchical States Machine (HSM) (Blanch et Beaudouin-Lafon 2006) ou les *machines à états hiérarchiques* sont une notation basée sur les automates à états finis.

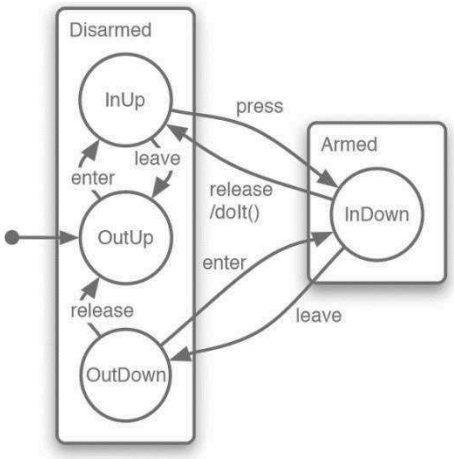


Figure 2.17. Représentation graphique en HSM du comportement d'un bouton (Blanch et Beaudouin-Lafon 2006)

La Figure 2.17 présente la représentation graphique en HSM du comportement d'un bouton. Cet exemple permet de mettre en évidence les différents éléments de la notation : les états, les variables et les transitions. La Figure 2.18 présente le code correspondant à ce comportement en utilisant la syntaxe, proposée par l'outil accompagnant la notation : HsmTk (pour HSM Toolkit). Cette syntaxe correspond à une extension du langage C++.

Les Hierarchical state machines ont comme avantages la facilité d'intégration dans une application existante et de permettre la modélisation de l'ensemble des états du système. Cependant, leur fondement sur les automates à états finis pose le problème du passage à l'échelle. En effet, il est difficile de modéliser les systèmes avec un grand nombre d'états car ceux-ci mènent à une explosion du nombre d'états. Ce plus, cette notation ne permet pas de modéliser les comportements concurrents.

```
01 hsm Button {
02    hsm Disarmed {
03       // variables
04       var svg::SVGElement *elem;
05       var svg::SVGGElement *armed = 0;
06       var svg::SVGGElement *disarmed = 0;
07
08       // initialization
09       init {
10          hsm::svg::Childs(elem).assert(armed)
11                                .assert(disarmed);
12          elem->removeChild(armed);
13       }
14
15       // visual aspect coherence
16       enter { elem->replaceChild(armed, disarmed); }
17       leave { elem->replaceChild(disarmed, armed); }
18
19       hsm OutUp {
20          - enter() > InUp
21       }
22
23       hsm InUp {
24          - leave() > OutUp
25          - press() > Armed::InDown
26       }
27
28       hsm OutDown {
29          - enter()   > Armed::InDown
30          - release() > OutUp
31       }
32    }
33
34    hsm Armed {
35       hsm InDown {
36          - leave() > Disarmed::OutDown
37
38          // action invocation
39          - release() { doIt(); } > Disarmed::InUp
40       }
41    }
42 }
```

Figure 2.18. Version textuelle du modèle HSM du comportement d'un bouton (Blanch et Beaudouin-Lafon 2006)

2.5.4 Notations basées sur les réseaux de Petri

Les réseaux de Petri (Petri 1962) sont une notation formelle permettant de modéliser le comportement d'un système à travers l'utilisation de places pouvant contenir des jetons et par un ensemble de transitions permettant les changements d'états. L'état du système est décrit par la distribution et la valeur des jetons dans ses différentes places.

Parmi les nombreuses notations basées sur les réseaux de Petri, certaines ont été utilisées ou spécifiées expressément pour la description comportementale de systèmes interactifs. Nous pouvons par exemple citer *ICO* (D. Navarre, P. Palanque, et al. 2009) que nous présentons ci-dessous, *OSU* (Keh et Lewis 1991), *MIML* (Latoschik 2002) ou les *réseaux de Petri colorés* (Rieder, Raposo et Pinho 2010).

Exemple d'ICO

Les ICO (pour Interactive Cooperative Objects) (Palanque 1992) sont une notation s'appuyant sur l'utilisation des réseaux de Petri à objets.

La Figure 2.19 présente la modélisation en ICO de la gestion de la sélection d'objets graphiques en ICO. Cet exemple permet de mettre en évidence un certain nombre d'éléments de la notation ICO : les places (par exemple, `MouseOnObject`), les jetons pouvant contenir des objets et donc des variables (par exemple, le jeton contenu dans la place MouseOnObject contenant les variables suivantes : `o,x,y,source,pres`), les transitions (par exemple, mouseMove_T1) et les arcs permettant de relier les places aux transitions et inversement).

L'avantage de cette notation est le fait qu'elle permette, du fait de son fondement sur les réseaux de Petri, de factoriser la description de systèmes complexes. Elles permettent, en plus de la modélisation de l'état du système, de décrire des comportements concurrents. Enfin, leur fondement sur les réseaux de Petri permet de mettre en place des méthodes d'analyse et de vérification formelles de plusieurs propriétés sur les systèmes interactifs.

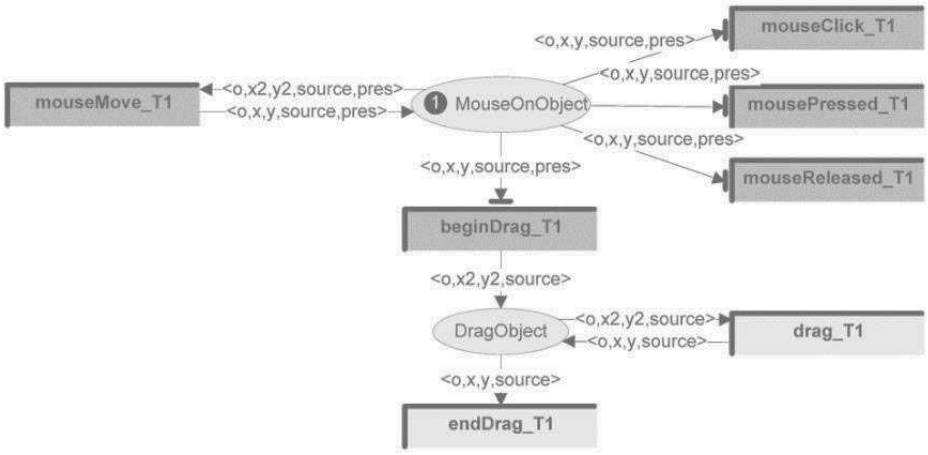


Figure 2.19. Modélisation de la gestion de la sélection d'objets graphiques en ICO (D. Navarre, P. Palanque, et al. 2009)

2.5.5 Synthèse

Nous venons de présenter les différents types de notations pour décrire le comportement des systèmes interactifs ainsi que leurs avantages et inconvénients.

La sélection d'une notation de description des systèmes interactifs critiques nécessite la complétion de plusieurs critères :

- *La couverture de l'interaction* : la notation doit permettre de couvrir tous les niveaux de l'interaction. Nous retenons ainsi, en nous basant sur le modèle architectural ARCH tous les composants autres que ceux ayant attraits au noyau fonctionnel : le contrôleur de dialogue, le composant d'interaction logique et le composant d'interaction physique. Pour qu'une notation permette de décrire complètement l'interaction entre l'utilisateur et le noyau fonctionnel, elle doit donc couvrir ces trois aspects.
- *Le passage à l'échelle* : la notation doit permettre de décrire des systèmes interactifs de taille réelle, de manière à pouvoir être appliquée à des études de cas industrielles telles que les systèmes interactifs des cockpits interactifs.
- *L'outillage* : la notation doit être supportée par un outil afin de rendre possible la description de systèmes complexes tels que les systèmes interactifs des cockpits interactifs.
- *L'expressivité* : la notation doit permettre de décrire de manière complète et non ambigüe tous les états possibles du système interactif ainsi que les possibilités pour les changements d'état. Elle doit permettre de prendre en compte les aspects spécifiques aux systèmes interactifs parmi lesquels nous retiendrons tout particulièrement les suivants : la description des objets et de leur valeur, la description des états, la représentation des événements, la représentation des aspects temporels ainsi que la représentation des comportements concurrents et de l'instanciation dynamique.

Tous ces critères nous permettent de choisir la notation ICO car elle seule permet de répondre à chacun d'entre eux. Ce choix est confirmé par l'utilisation de cette notation pour le développement de systèmes interactifs critiques dans de nombreux travaux. Elle a ainsi été proposée pour le développement de tels systèmes dans un processus spécifiques aux systèmes interactifs critiques permettant de garantir les aspects utilisabilité et fiabilité (Martinie, Palanque et Navarre, et al. 2012). Elle a également été proposée pour la description de systèmes interactifs critiques comme les applications des cockpits interactifs (Barboni, Palanque, et al. 2007), les applications post-WIMP appliquées au contexte des cockpits interactifs (Hamon, Palanque, et al. 2013) ainsi que pour l'introduction de mécanismes de tolérance aux fautes dans les applications des cockpits interactifs (Tankeu-Choitat, et al. 2011).

2.6 La tolérance aux fautes

2.6.1 Principes généraux de la tolérance aux fautes

La tolérance aux fautes est le seul moyen d'augmenter le niveau de sûreté de fonctionnement d'un système par rapport aux fautes naturelles en opération car celles-ci sont imprévisibles et inévitables. Dans le domaine des systèmes critiques, la tolérance aux fautes fait l'objet de plusieurs travaux visant à développer des techniques, mécanismes et architectures permettant d'accomplir la détection des erreurs et leur recouvrement. Ces travaux reposent sur trois principes fondamentaux :

- *La redondance* qui consiste à déployer plusieurs composants (matériel et/ou logiciel) responsables de la même fonctionnalité et qui permet la tolérance des fautes naturelles.
- *La diversification* qui consiste à déployer plusieurs composants (matériel et/ou logiciel) responsables de la même fonctionnalité, issus des mêmes spécifications mais de développement différents. Ces différents composants sont appelés des *variantes*. Les diversifications matérielle et logicielle permettent de tolérer respectivement les fautes matérielles de développement et les fautes logicielles de développement. La diversification est très contraignante : elle suppose de développer de manière différente plusieurs variantes d'un même composant à partir d'une spécification commune. Cela inclut un développement par des équipes différentes et issues de formations différentes, utilisant des algorithmes, des langages et des outils de programmation différents.
- *La ségrégation* qui consiste à une isolation spatiale et temporelle des composants répliqués et/ou diversifiés telle que celle définie par le standard ARINC specification 653 (AEEC 2003) (voir section 3.2.3).

Pour choisir un mécanisme de tolérance aux fautes adapté à un système parmi le large choix proposé dans la littérature, il est nécessaire de mettre en place des critères de sélection. Dans ce but, (Stoicescu, Fabre et Roy 2011) et (Stoicescu 2013) ont classés les mécanismes de tolérance aux fautes les plus connus et leur raffinement en fonction de deux critères :

- *Leur capacité de tolérance aux fautes* : quel types de fautes ils permettent de traiter (fautes logicielles, matérielles, permanentes, temporaires, …).
- *Les spécificités du système concerné* : est-il déterministe, nécessite-t-il une sauvegarde de son état et si oui, cet état est-il accessible ?

Ainsi, pour sélectionner un mécanisme de tolérance aux fautes pour un système particulier, il s'agit de définir son modèle de faute, c'est-à-dire d'établir la liste des fautes susceptibles d'affecter le système et d'étudier le système concerné de manière à déterminer s'il est déterministe ou pas, s'il nécessite une sauvegarde de son état et si cet état est accessible ou non.

2.6.2 Principales architectures de tolérance aux fautes

La Figure 2.20 est extraite des travaux de (Stoicescu, Fabre et Roy 2011) et présente une classification des principaux mécanismes de tolérance aux fautes en fonction des deux critères identifiés précédemment. Nous présentons dans les sous-sections suivantes les principaux mécanismes de tolérance aux fautes existants. Il est important de noter que les mécanismes permettant la tolérance des fautes matérielles et ceux permettant la tolérance des fautes logicielles présentent des architectures logicielles similaires basées sur la redondance des composants. La différence principale entre ces deux types de mécanismes de tolérance est que la tolérance des fautes logicielles nécessite la diversification de ces composants alors que la tolérance aux fautes naturelles nécessite leur ségrégation. En appliquant à ces architectures logicielles à la fois la diversification et la ségrégation des composants il est possible de couvrir à la fois les fautes logicielles et matérielles. Ainsi, bien que nous présentions dans les sous-sections suivantes les architectures logicielles correspondantes aux mécanismes de tolérance aux fautes logicielles, celles-ci sont similaires aux architectures correspondantes aux mécanismes de tolérance aux fautes matérielles.

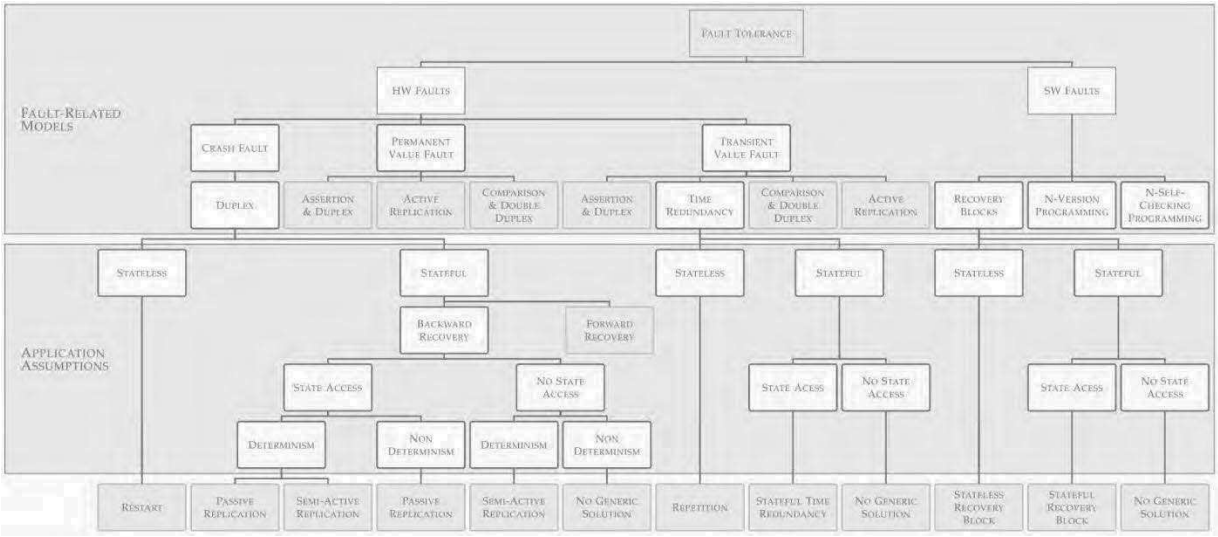


Figure 2.20. Classification des mécanismes de tolérance aux fautes (Stoicescu 2013)

Blocs de recouvrement

Le principe des blocs de recouvrement est présenté en Figure 2.21. Ainsi, un composant implémentant l'approche des blocs de recouvrement est composé de N versions du composant (appelés des alternants) et d'un test d'acceptation. Si la première version du composant fournit un résultat correct (conforme au test d'acceptation), c'est ce résultat qui est envoyé comme sortie des blocs de recouvrement. Si le résultat de la première version du composant n'est pas conforme au test d'acceptation, on reporte l'exécution du calcul à la deuxième version du composant, et ainsi de suite jusqu'à ce qu'un composant fournisse un résultat compatible avec le test d'acceptation ou jusqu'à épuisement des versions (dans ce cas, le composant est globalement défaillant). En cas d'erreur d'une ou plusieurs version(s), le composant ne pourra fournir de résultat qu'après l'exécution consécutive du calcul sur une ou plusieurs version(s), impliquant ainsi une suspension de service pouvant se révéler longue.

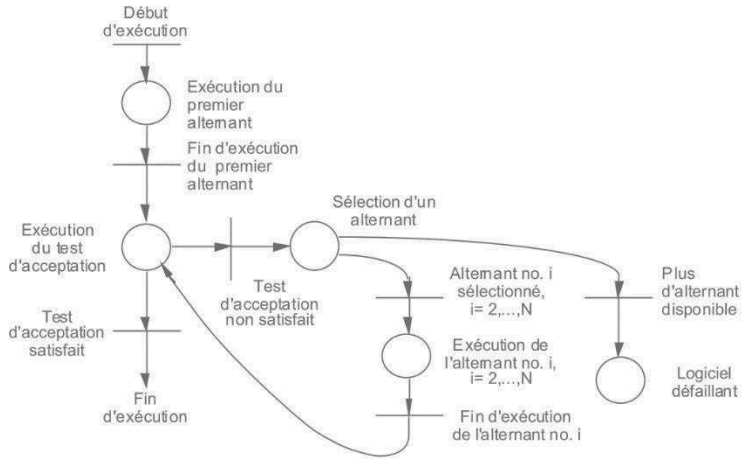


Figure 2.21. Modèle d'exécution des blocs de recouvrement (Laprie, Deswartes, et al. 1996)

Les blocs de recouvrement ou Recovery Blocks (Randell 1975) ont une architecture logicielle similaire à celle de la redondance temporelle ou Time Redundancy.

Programmation en n-versions (NVP)

La programmation en n-versions (Avizienis 1985) se base sur la création de n versions du composant auquel on souhaite appliquer l'approche de tolérance aux fautes. À ces n versions, s'ajoutent deux composants supplémentaires : un composant permettant la distribution des entrées aux n versions et un composant permettant d'effectuer un vote sur leurs sorties. Ainsi, l'architecture logicielle d'un

composant NVP correspond à celle présentée en Figure 2.22. Plus précisément, lorsque le composant NVP reçoit une entrée, le composant de distribution (`Dispatch`) transmet cette entrée aux n versions qui exécutent leur calcul en parallèle et qui fournissent leur résultat au composant chargé du vote (`Vote`). Celui-ci effectue alors un vote majoritaire sur les résultats fournis par les n versions et renvoie le résultat majoritaire (supposé juste) comme sortie du composant NVP.

Comme expliqué précédemment, cette architecture logicielle est similaire à celle de l'approche de réplication active. Ainsi, si les n versions du composant présenté en Figure 2.22 sont remplacées par une seule et unique version, redondée et ségréguée, le composant ne sera pas capable de détecter ou recouvrir les erreurs dues aux fautes logicielles mais il pourra tolérer les erreurs dues aux fautes matérielles.

L'architecture logicielle de la programmation en N-versions ou N-Version-Programming (NVP) est similaire à celle de la réplication active ou Active Replication (Chereque, et al. 1992).

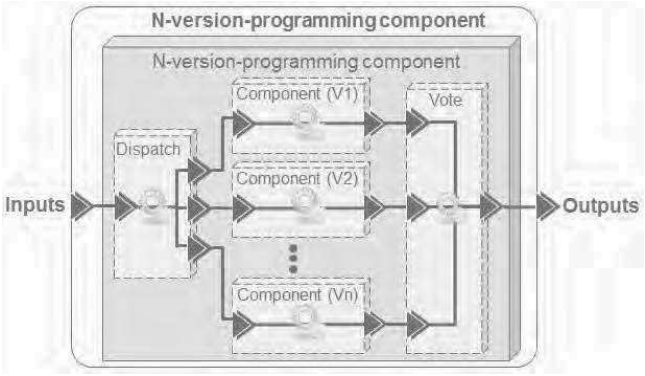


Figure 2.22. Architecture logicielle d'un composant NVP

Composants autotestables (SC)

Un composant autotestable (Laprie, Deswartes, et al. 1996) est un composant (appelé composant fonctionnel ou COM pour commande) auquel a été adjoint un composant de contrôle (appelé MON pour moniteur) capable de vérifier le comportement du composant fonctionnel et de notifier une erreur en cas d'incohérence. L'architecture logicielle d'un tel composant est présentée en Figure 2.23. On peut remarquer sur cette figure un composant supplémentaire : le composant de distribution (`Dispatch`) qui permet la distribution des entrées aux deux composants (le COM et le MON). Un composant autotestable permet la détection des erreurs mais ne permet pas à lui tout seul leur recouvrement, c'est pour cela qu'a été introduite l'approche de la programmation n-autotestable (Laprie, Arlat, et al. 1990).

L'architecture autotestable ou self-checking correspond au fondement de la programmation n-autotestable et est également la plus fréquemment utilisée pour la détection des erreurs matérielles.

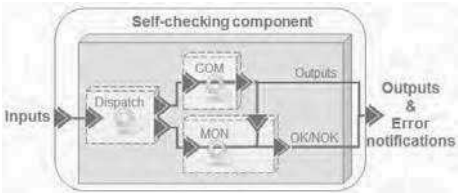


Figure 2.23. Architecture logicielle d'un composant autotestable

Programmation n-autotestable (NSCP)

Un composant n-autotestable (Laprie, Arlat, et al. 1990) est un composant avec une architecture logicelle basée sur la redondance de composants autotestables. Ainsi, un composant n-autotestable est composé, comme présenté Figure 2.24, de n composants autotestables, d'un composant permettant la distribution des entrées et d'un composant de décision. Le principe est que le composant de distribution (Dispatch) distribue les entrées aux différents composants autotestables (autrement dit aux différents composants COM et MON de manière à ne pas multiplier les composants de distribution). Les différents composants autotestables traitent ces entrées en parallèle et retournent au composant de décision leurs

résultats ainsi qu'une notification d'erreur ou de validité de ce résultat. Si le premier composant a exécuté son calcul correctement, le composant de décision choisira ce résultat comme résultat du composant n-autotestable ; si le premier composant autotestable a notifié une erreur sur son résultat, le composant de décision s'intéressera au résultat du deuxième composant autotestable et ainsi de suite jusqu'à ce qu'un composant autotestable fournisse un résultat sans erreur ou jusqu'à épuisement des composants autotestables (dans ce cas, le composant enverra une erreur).

L'archtecture logicielle de la programmation n-autotestable est similaire à celle de la comparaison avec double duplex (`Comparison & Double Duplex`).

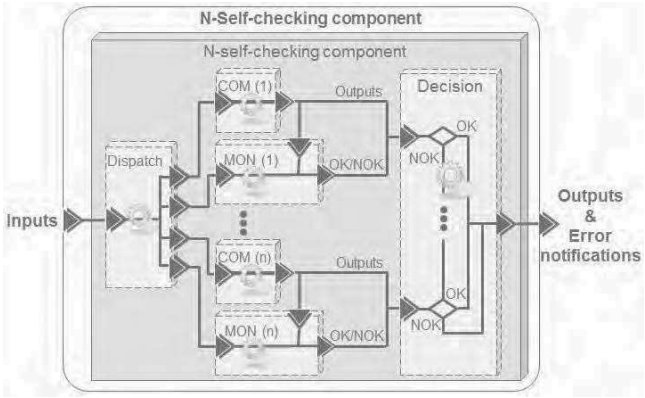


Figure 2.24. Architecture logicielle d'un composant NSCP

Exemples d'utilisation de la tolérance aux fautes pour les systèmes avioniques

La tolérance aux fautes est très présente dans les systèmes avioniques et notamment dans les systèmes de commandes de vols électriques. Nous présentons ci-dessous deux exemples notoires d'utilisation de mécanismes de tolérance aux fautes dans l'aéronautique à travers les exemples des commandes de vol électriques Airbus et Boeing.

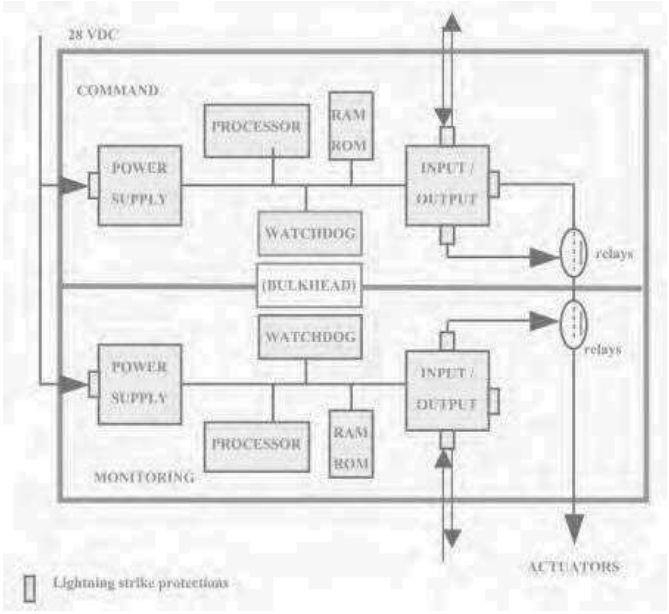


Figure 2.25. Architecture simplifiée d'un calculateur de commande de vol Airbus (Traverse, Lacaze et Souyris 2004)

Les commandes de vol chez Airbus reposent sur une programmation n-autotestable (Traverse, Lacaze et Souyris 2004). Chaque calculateur est composé, comme présenté en Figure 2.25, de deux calculateurs indépendants et ségrégués : le `COM` (pour `Command` ou commande) qui effectue la fonction allouée au calculateur et le `MON` (pour `Monitoring` ou contrôle) qui vérifie le fonctionnement du calculateur `COM`. De plus, ces calculateurs sont redondants sur le principe de la programmation n-

autotestable. On peut ainsi retrouver 3 calculateurs primaires (PRIM) et 3 calculateurs secondaires (SEC). Les calculateurs SEC utilisent un matériel différent des calculateurs PRIM et un composant logiciel diversifié grâce à l'utilisation de lois de commande plus simples.

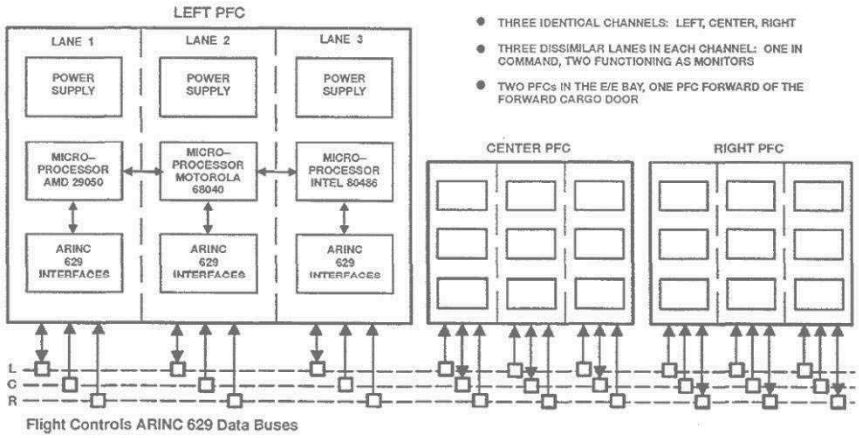


Figure 2.26. . Architecture simplifiée d'un calculateur de commande de vol Boeing (Y. .. Yeh 1996)

Les commandes de vol chez Boeing reposent sur une programmation en N-Versions (Y. .. Yeh 1996). Ainsi les commandes de vol chez Boeing, comme présenté en Figure 2.26 sont constituées de trois calculateurs numériques centraux similaires appelés PFC (*Primary Flight Control*). Chaque PFC est lui-même constitué de trois calculateurs dissimilaires autant sur le plan matériel que logiciel.

2.7 Synthèse

Ce chapitre a présenté les caractéristiques et les propriétés des systèmes interactifs et des systèmes critiques. Ceci nous a permis de caractériser les systèmes faisant l'objet de cette thèse : les systèmes interactifs critiques et de mettre en valeur les problématiques liées à ces systèmes. Nous avons ainsi pu déterminer les deux propriétés fondamentales de ces systèmes : la sûreté de fonctionnement et l'utilisabilité ainsi que les conflits que ces deux propriétés peuvent impliquer.

Nous avons ensuite présenté, en nous appuyant sur une classification de toutes les fautes pouvant affecter les systèmes interactifs, les différentes méthodes, outils et techniques pour la prise en compte de ces différentes fautes.

Pour répondre aux problématiques de description formelle des systèmes interactifs, nous avons présenté les différentes architectures logicielles spécialisées pour les systèmes interactifs ainsi que les différentes notations permettant la description comportementale des systèmes interactifs. Cette présentation nous a amené au choix d'un modèle architectural et d'une notation formelle sur lesquels fonder nos travaux : le modèle architectural ARCH et la notation formelle ICO.

Enfin, pour répondre à la problématique de tolérance aux fautes, nous avons présenté les différentes techniques et architectures pour la tolérance aux fautes.

Chapitre 3. Contexte applicatif : les cockpits interactifs

Sommaire

Ce chapitre décrit le contexte des systèmes interactifs dans les cockpits afin de mettre en évidence les contraintes et les exigences que ceux-ci impliquent. Il présente ainsi les concepts que nous devrons respecter tout au long de cette thèse afin de garantir que l'architecture et l'approche que nous proposons puissent être appliquées aux systèmes interactifs dans les cockpits.

Nous présentons dans un premier temps, grâce à l'exemple du cockpit interactif de l'A380, les architectures logicielles et physiques des cockpits interactifs qui lui sont similaires. Ceci inclut notamment la présentation du standard ARINC 661 qui définit les applications interactives de tels cockpits.

Nous présentons dans un second temps les contraintes de développement logiciel dans l'avionique liées au besoin de certification de l'avion, et plus particulièrement celles qui s'appliquent au développement logiciel et aux systèmes interactifs.

Enfin, nous présentons l'état actuel de sûreté de fonctionnement des différents systèmes de commande et contrôle dans les cockpits interactifs ainsi que les exigences souhaitées pour pouvoir commander et contrôler des systèmes avioniques critiques à l'aide des systèmes interactifs.

3.1 Les cockpits interactifs

Le système interactif dans le cockpit est appelé CDS (Control and Display System). Il a pour but de fournir à l'équipage les informations et les moyens d'interaction nécessaires à certaines tâches de pilotage de l'avion. L'architecture matérielle et logicielle du CDS est différente en fonction de l'avion dans lequel elle est implémentée. Pour pouvoir donner des informations concrètes et illustratives, nous détaillons ici l'exemple du cockpit interactif de l'Airbus A380 (voir Figure 3.1) qui est représentatif, de par son architecture matérielle et logicielle, de l'ensemble des cockpits interactifs. De plus, le cockpit interactif de l'A380 est d'autant plus intéressant qu'il s'agit du premier système d'affichage interactif dans l'aviation commerciale.

3.1.1 Architecture matérielle

Le CDS de l'A380, représenté en Figure 3.1, est composé de 8 dispositifs d'affichage appelés DU (Display Units). Ces dispositifs d'affichage présentent l'état de certains systèmes avioniques ainsi que des applications interactives avec lesquelles les pilotes peuvent interagir via deux dispositifs de contrôle appelés KCCU (Keyboard and Cursor Control Unit). Les KCCU (voir Figure 3.2), sont des périphériques d'entrée regroupant dans le même équipement un clavier et une trackball. Les DU (voir

Figure 3.3), sont plus qu'un simple dispositif d'affichage, ce sont des calculateurs complexes assimilables à des ordinateurs de bureau. En effet, une DU est constituée, d'un écran LCD, d'une carte graphique et d'un module d'acquisition et de traitement. La carte graphique est responsable des affichages sur l'écran. Le module d'acquisition et de traitement est comparable au processeur d'un ordinateur et est responsable du système d'exploitation, des applications, de la mémoire ainsi que de la gestion des entrées et sorties.

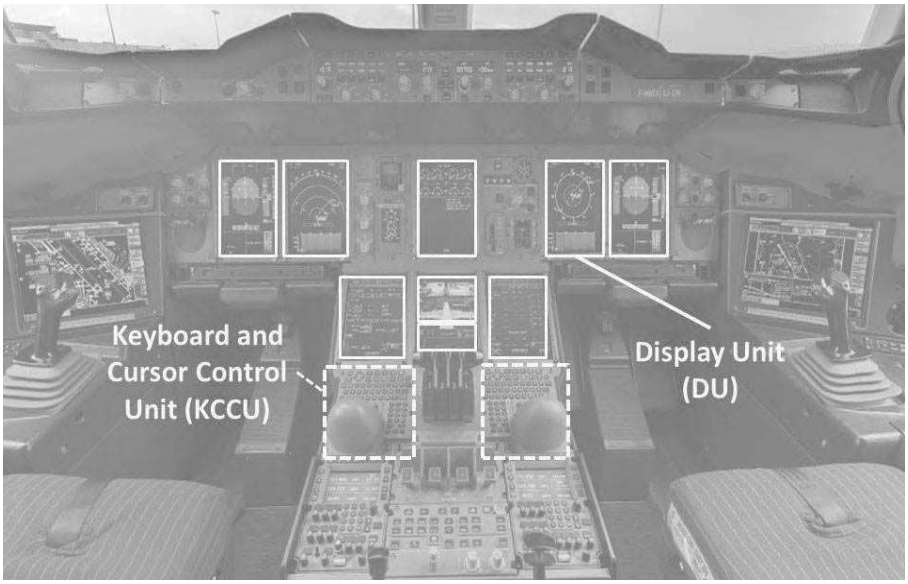


Figure 3.1. Cockpit interactif de l'Airbus A380

Figure 3.2. Exemple d'un KCCU (Keyboard and Cursor Control Unit)

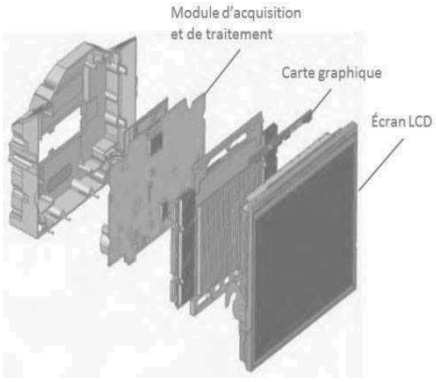


Figure 3.3. Architecture matérielle d'une DU (Display Unit)

Les 8 DUs du cockpit de l'A380 ne présentent pas les mêmes informations et ne sont pas destinées à la même utilisation. Ainsi, comme présenté en Figure 3.4, les DUs du côté gauche (L1, L2 et L3) sont destinées au pilote qui peut interagir avec elles grâce au KCCU de gauche. De la même manière, les DUs de droite (R1, R2 et R3) sont destinées au co-pilote qui peut interagir avec elles grâce au KCCU de droite. Les DUs centrales (C1 et C2), quand à elles, sont partagées par le pilote et le co-pilote.

Les DU sont connectées entre elles et avec les KCCU via 2 réseaux ségrégués et redondants de type CAN (Controller Area Network, (International Standard Organization 2009)) (voir Figure 3.4). Il est ainsi possible de distinguer les réseaux redondants CAN 1.1 et CAN 1.2 pour la connexion des DU et du KCCU du pilote et les réseaux redondés CAN 2.1 et 2.2 pour la connexion entre les DU et le KCCU du co-pilote. Le CDS (composé des 8 DUs et des 2 KCCUs) est connecté avec les systèmes avioniques via un réseau AFDX (Avionics Full DupleX switched Ethernet (AEEC, ARINC 664-P7 2009)).

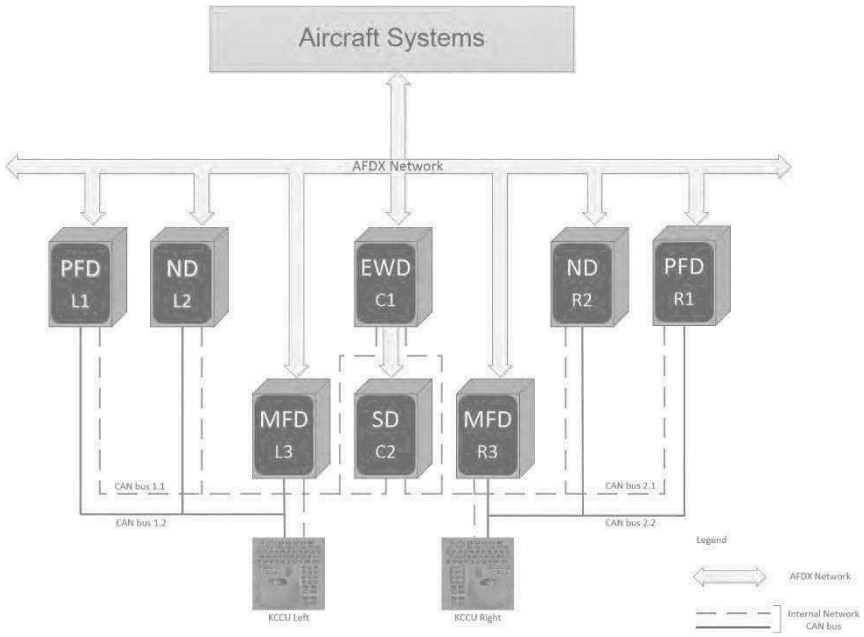


Figure 3.4. Architecture matérielle du CDS dans l'Airbus A380

Les DU permettent d'afficher des informations provenant de différents composants avioniques. L'ensemble des informations affichées sur une DU est appelé un format. Les formats permettent ainsi de grouper toutes les informations nécessaires aux pilotes pour une tâche donnée, on peut donc parler de regroupement fonctionnel. La configuration nominale de la répartition des différents formats sur les 8 DU du CDS de l'A380 est présentée en Figure 3.4. Cette configuration peut varier : des reconfigurations sont possibles en cas d'avaries des DU présentant les formats indispensables pour les pilotes (par exemple, le Primary Flight Display). Tous les formats disponibles dans le CDS de l'A380 sont visibles en Figure 3.4, nous distinguons ainsi :

- *Le PFD (Primary Flight Displays)* qui permet d'afficher les paramètres de vol primaires, tels que la vitesse et l'altitude, permettant le pilotage de l'avion à court terme.
- *Le ND (Navigation Displays)* qui permet d'afficher les paramètres pour la navigation de l'avion tels que les informations météorologiques, le plan de vol, les autres avions dans le champ de vol, les aéroports et balises à proximité de l'avion.
- *Le E/WD (Engine Warning Display)* qui permet d'afficher les paramètres moteurs et les messages d'alerte.
- *Le SD (System Display)* qui permet d'afficher les paramètres d'état de l'ensemble des systèmes de l'avion tels que les systèmes de conditionnement d'air, de ventilation, de pressurisation, et hydrauliques.
- *Le MFD (Multi Function Display)* qui permet d'afficher différentes fonctions de l'avion telles que le contrôle du trafic aérien, le système de surveillance et la gestion du plan de vol.

3.1.2 Architecture logicielle

Les échanges entre le CDS et les systèmes avioniques ainsi que l'architecture logicielle des applications interactives dans les DU dépendent du standard ARINC specification 661 (AEEC 2013), plus communément appelé ARINC 661 ou encore A661.

3.1.2.1 Le standard ARINC specification 661

Le standard ARINC specification 661 (AEEC 2013) a été créé afin de standardiser les systèmes de contrôle et d'affichage (CDS) dans les avions. Il a été rédigé par les principaux acteurs de l'aéronautique tels que Thalès, Airbus, Boeing, Rockwell Collins, Honeywell, … et adopté en 2001 par l'AEEC (Airlines Electronic Engineering Committe). Les objectifs principaux de ce standard sont les suivants :

- Permettre l'introduction de l'interactivité dans le CDS en mettant en place une base pour standardiser l'interface homme-machine dans le cockpit et ainsi rendre les dispositifs plus faciles à utiliser et plus efficaces en permettant l'affichage d'informations complexes dans les systèmes du bord.
- Minimiser les coûts de développement des systèmes d'affichage.
- Minimiser les coûts d'ajout ou de modification d'une nouvelle fonction d'affichage pendant la durée de vie d'un avion (une trentaine d'années pour les avions commerciaux).
- Minimiser le coût de gestion de l'obsolescence des composants matériels dans un environnement où les évolutions technologiques sont rapides.

Pour cela, le standard ARINC 661 définit un protocole de communication entre le CDS et les parties logicielles des systèmes avioniques, plus communément appelées UA (pour User Applications). Cette communication repose sur la manipulation d'objets graphiques interactifs appelés *widgets* dont l'interface logicielle est définie par le standard.

3.1.2.2 Architecture logicielle d'une application suivant le standard ARINC 661

Les applications respectant le standard ARINC 661 doivent répondre à une certaine architecture logicielle. Elles suivent ainsi un concept de fenêtrage semblable à celui des ordinateurs de bureau (voir Figure 3.5). Le dispositif physique sur lequel se fait l'affichage est appelé Display Unit (DU). La surface de l'écran peut être divisée en plusieurs fenêtres, chacune d'elle pouvant être elle-même divisée en un ou plusieurs layers. Un *layer* correspond au plus haut niveau de la hiérarchie de widgets interagissant avec un système avion à travers son interface logicielle : l'UA.

Les *widgets* sont les composants interactifs de base définis par le standard ARINC 661. Ils sont assez semblables aux composants interactifs avec lesquels nous avons l'habitude d'interagir sur nos ordinateurs de bureau. La Figure 3.6 présente quelques exemples de ces widgets :

- Le *PicturePushButton*, un bouton permettant au pilote de déclencher des commandes (*e.g.* l'engagement d'un niveau de vol),
- L'*EditBoxNumeric*, dédiée à la saisie de valeurs numériques (*e.g.* la valeur d'un niveau de vol),
- Les *CheckButtons* permettant la sélection d'une option parmi un ensemble d'options disponibles.

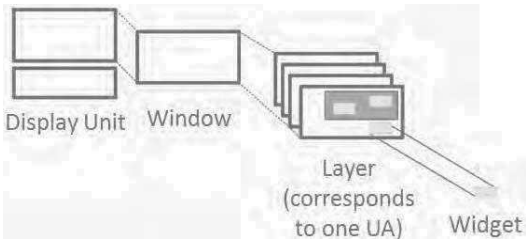


Figure 3.5. Principe de fenêtrage du standard ARINC 661 (AEEC 2013, 9)

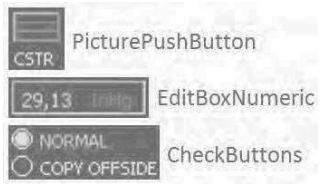


Figure 3.6. Exemples de widgets

Pour une même application les widgets sont organisés entre eux de manière hiérarchique. Le composant en charge de leur organisation est appelé le *graphe de scène*. Il peut être comparé à un arbre de widgets. En effet, certains widgets, les *containers*, ont la capacité de regrouper d'autres widgets. Les containers correspondent aux nœuds de l'arbre et sont appelés *parents* tandis que les widgets qu'ils contiennent correspondent sont appelés *enfants*. Le graphe de scène permet de structurer l'application :

- *De manière géographique :* la position des widgets sur l'écran dépend de celle de leurs parents ;
- *De manière fonctionnelle :* certaines propriétés des widgets, telles que leur visibilité ou leur activation, sont propagées par les containers vers leurs enfants.

Les widgets correspondant à une fonction peuvent donc être regroupés dans un seul parent. Cette organisation permet à l'UA d'activer ou de désactiver en une seule action tous les widgets correspondant à cette fonction ainsi que de les regrouper géographiquement dans un même espace sur l'écran.

Le graphe de scène correspondant à une UA est défini préalablement dans un fichier de définition appelé User Application Definition File (ou UADF) qui contient l'ensemble des layers de cette UA. L'UADF affecte un identifiant unique à chaque widget. Ce fichier est connu à la fois de l'UA et du CDS et assure la cohérence structurelle entre ces deux éléments du système interactif : il permet l'instanciation des widgets lors de l'initialisation du CDS et fournit la connaissance des widgets disponibles et leur organisation à l'UA.

3.1.2.3 Description des widgets dans le standard ARINC 661

Plus que le protocole permettant la communication entre le CDS et l'UA, le standard ARINC 661 définit complètement l'interface logicielle et fonctionnelle des widgets car ils sont la base de l'interaction. Lors de sa création en 2001, le standard ARINC 661 définissait 41 widgets. Actuellement, début 2015, le standard en est à sa cinquième révision et définit 77 widgets.

Pour chacun de ces widgets, le standard ARINC 661 définit les différents paramètres les constituant (leur valeur peut être fixée lors de l'initialisation ou modifiable par l'UA pendant l'exécution par le biais des méthodes appelées `A661_setParameters`) ; il définit également les événements que les widgets sont susceptibles d'envoyer à l'UA (appelés `A661_Events`). Toutefois, le standard ne définit par leur « Look and Feel », autrement dit, il ne définit ni leurs caractéristiques graphiques (*Look*), ni leur comportement (*Feel*).

Concrètement, la définition de chaque widget dans le standard est divisée en cinq parties :

- Une description textuelle du widget et de son comportement (`Definition section`) ;
- Une table de description des paramètres du widget (`Widget parameter table`) ;
- Une table de description des paramètres nécessaires à l'instanciation du widget (`Creation structure table`) ;
- Une description des événements pouvant être envoyés par le widget (`Event structure table(s)`) ;
- Une description des paramètres modifiables à l'exécution par l'UA (`Run-time modifiable parameters table`).

Pour illustrer la description des widgets dans le standard ARINC 661, nous prenons l'exemple du *PicturePushButton* qui est un bouton poussoir permettant à l'utilisateur de lancer une action et permettant l'affichage d'une image. Sur une application interactive classique telle qu'un traitement de texte, nous pourrions donner l'exemple du bouton représenté par une image de disquette et permettant l'enregistrement du document :  . Sa définition par le standard ARINC 661 est présentée en Figure 3.7.

3.3.28 PicturePushButton

Categories:

- Interactive
- Graphical representation
- Text string

Description:
A PicturePushButton widget is a PushButton including a picture and possibly a string.

Restriction:
None

Figure 3.7. Description du *PicturePushButton* (extrait du standard ARINC 661-5 (AEEC 2013, 138))

La table de description des paramètres du *PicturePushButton* est représentée en Figure 3.8. Elle est divisée en deux parties : les paramètres communs (utilisés par la plupart des widgets) et les paramètres spécifiques au *PicturePushButton*. Chaque paramètre est défini par :

- Son nom (colonne `Parameters`),
- La possibilité de modification de sa valeur (colonne `Change`), qui peut être :
    - Fixée à l'instanciation du widget sans possibilité de modification(`D`) ;

- Nécessaire à l'instanciation du widget et modifiable à l'exécution (`DR`) ;
- Seulement modifiable à l'exécution (`R`).

- Sa description (colonne `Description`).

Table 3.3.28-1 – PicturePushButton Parameters

Parameters	Change	Description
Commonly used parameters		
WidgetType	D	A661_PICTURE_PUSH_BUTTON
WidgetIdent	D	Unique identifier of the widget
ParentIdent	D	Identifier of the immediate container of the widget
Visible	DR	Visibility of the widget
Enable	DR	Ability of the widget to be interactive
StyleSet	DR	Reference to predefined graphical characteristics inside CDS
PosX	D	The X position of the widget reference point
PosY	D	The Y position of the widget reference point
SizeX	D	The X dimension size (width) of the widget
SizeY	D	The Y dimension size (height) of the widget
NextFocusedWidget	DR	Widget ident of next widget to be focused. See Section 3.1.3.5.
AutomaticFocusMotion	DR	Automatic motion of the focus on widget specified in NextFocusedWidget parameter
Specific parameters		
MaxStringLength	D	Maximum length of the label text
Alignment	D	Alignment of the text within the label area of the widget LEFT RIGHT CENTER
LabelString	DR	Label of the PicturePushButton
Picture Reference	DR	Reference of the picture
PicturePosition	D	The string position depends on the picture position: CENTER LEFT RIGHT TOP BOTTOM
EntryValidation	R	Indicator notifying the CDS that the UA has completed processing the entry or selection event. This flag also indicates the results of that processing. A661_FALSE A661_TRUE

Figure 3.8. Description des paramètres du *PicturePushButton* (extrait du standard ARINC 661-5 (AEEC 2013, 139))

La table d'instanciation du *PicturePushButton* est présentée en Figure 3.9. Cette table décrit, pour chaque paramètre nécessaire à l'instanciation du widget, son type, sa taille et d'éventuelles restrictions sur sa valeur.

Table 3.3.28-2 – Picture PushButton Creation Structure

CreateParameterBuffer	Type	Size (bits)	Value/Range When Necessary
WidgetType	ushort	16	A661_PICTURE_PUSH_BUTTON
WidgetIdent	ushort	16	
ParentIdent	ushort	16	
Enable	uchar	8	A661_FALSE A661_TRUE A661_TRUE_WITH_VALIDATION
Visible	uchar	8	A661_FALSE A661_TRUE
PosX	long	32	
PosY	long	32	
SizeX	ulong	32	
SizeY	ulong	32	
StyleSet	ushort	16	
NextFocusedWidget	ushort	16	
PictureReference	ushort	16	
MaxStringLength	ushort	16	
PicturePosition	uchar	8	A661_LEFT A661_CENTER A661_RIGHT A661_TOP A661_BOTTOM
AutomaticFocusMotion	uchar	8	A661_FALSE A661_TRUE
Alignment	uchar	8	A661_LEFT A661_CENTER A661_RIGHT
UnusedPad	N/A	8	0
LabelString	string	8 * string length + Pad	Followed by zero, one, two, or three extra NULL for alignment of 32 bits.

Figure 3.9. Table d'instanciation du *PicturePushButton* (extrait du standard ARINC 661-5 (AEEC 2013, 139))

La description des événements pouvant être envoyés par le *PicturePushButton* est présentée en Figure 3.10. La description des événements du *PicturePushButton* ne comprend qu'une seule table car celui-ci ne peut envoyer qu'un seul événement. Cette table décrit le nom de cet événement (`A661_EVT_SELECTION`) et les paramètres qu'il transmet (cet événement ne transmet aucun paramètre). Le *PicturePushButton* enverra cet événement à l'UA lorsqu'il aura reçu un clic provenant d'une action de l'utilisateur (à condition qu'il soit à cet instant dans le bon état interne : visible et actif).

Table 3.3.28-3 – Picture PushButton Event Structures: A661_EVT_SELECTION

EventStructure	Type	Size (bits)	Value/Description
EventIdent	ushort	16	A661_EVT_SELECTION
UnusedPad	N/A	16	0

Figure 3.10. Description des événements du *PicturePushButton* (extrait du standard ARINC 661-5 (AEEC 2013, 140))

La description des paramètres modifiables à l'exécution du *PicturePushButton* est présentée en Figure 3.11. Elle décrit, pour chaque paramètre, son type et son identifiant. Ces paramètres sont modifiés par le widget sur demande de l'UA. Pour fournir un tel service, le widget propose à l'UA des méthodes appelées `setParameters`. Ainsi, pour modifier le paramètre d'activation d'un *PicturePushButton* (appelé `Enable`), l'UA va appeler la méthode `setEnable()` du widget avec la valeur de son choix. De cette manière, l'UA va demander à un *PicturePushButton* de passer inactif par l'appel de méthode `setEnable(A661_FALSE)`. Lors de la réception de cet appel de méthode, le widget modifie la valeur de son paramètre, modifiant donc à la fois son état interne et son affichage graphique (pour notifier son inactivité au pilote, il sera par exemple grisé).

Table 3.3.28-4 – Picture PushButton Runtime Modifiable Parameters

Name of the Parameter to Set	Type	Size (bits)	ParameterIdent Used in the ParameterStructure
Enable	uchar	8	A661_ENABLE
Visible	uchar	8	A661_VISIBLE
LabelString	string	{32}+	A661_STRING
PictureReference	ushort	16	A661_PICTURE_REFERENCE
StyleSet	ushort	16	A661_STYLE_SET
EntryValidation	uchar	8	A661_ENTRY_VALID
NextFocusedWidget	ushort	16	A661_NEXT_FOCUSED_WIDGET
AutomaticFocusMotion	uchar	8	A661_AUTO_FOCUS_MOTION

Figure 3.11. Description des paramètres modifiables à l'exécution du PicturePushButton (extrait du standard ARINC 661-5 (AEEC 2013, 140))

La Figure 3.12 résume en deux diagrammes de séquence les échanges possibles entre le CDS et les systèmes de l'avion (notamment leur partie informatique : l'UA).

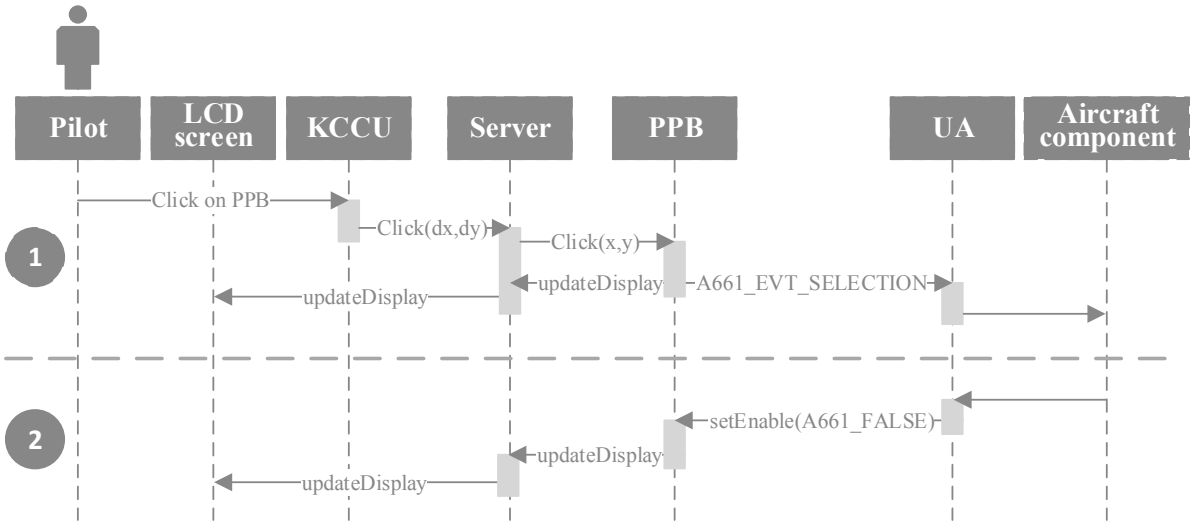


Figure 3.12. Diagrammes de séquence des échanges entre le CDS et l'UA (1) et entre l'UA et le CDS (2)

Dans la première partie de la figure (Figure 3.12-1), l'utilisateur décide d'engager un contrôle en cliquant, à l'aide du KCCU sur un *PicturePushButton* (ou PPB). Le KCCU envoie le clic au Server qui identifie le widget ciblé par ce clic (PPB) et lui transfère l'événement. Le *PicturePushButton* traite cet événement en envoyant un A661_EVT_SELECTION à l'UA et en demandant une modification de son rendu graphique au Server. Enfin, l'UA transmet au système avion (Aircraft component) le contrôle correspondant.

Dans la deuxième partie de la figure (Figure 3.12-2), le système avionique notifie une modification de son état qui nécessite une modification de l'interface graphique (ici, il s'agit de la désactivation d'un *PicturePushButton* de manière à notifier au pilote qu'il ne peut plus engager le contrôle correspondant). Dans ce cas, l'UA fait un appel de méthode setEnable(A661_FALSE) sur le *PicturePushButton* PPB. Le widget met alors à jour son état interne et notifie ce changement d'état au Server qui actualise son affichage (ici, le bouton sera grisé de manière à notifier à l'utilisateur son inactivité).

Le standard ARINC 661 propose une classification des widgets en six catégories en fonction de leurs fonctionnalités. Ces catégories ne sont pas exclusives et un widget peut appartenir à plusieurs catégories. Les travaux de (A. Tankeu-Choitat 2011) proposent une classification plus poussée des widgets définis par le standard ARINC 661. Celle-ci s'appuie sur des études ergonomiques et plus particulièrement sur les travaux de (Farenc 1997) qui ont permis de constater que certains widgets étaient concernés par les mêmes propriétés ergonomiques. La classification proposée par (A. Tankeu-Choitat 2011), présentée en Figure 3.13, est constituée de classes regroupant des widgets concernés par les mêmes règles ergonomiques. Les différentes classes définies forment une arborescence à plusieurs niveaux d'abstraction, les sous-classes héritant des règles ergonomiques des classes précédentes dans la hiérarchie.

Cette classification présente les avantages suivants (A. Tankeu-Choitat 2011) :

- Elle permet de suivre l'évolution du standard ARINC 661 en proposant une vue organisée de l'ensemble des widgets du standard.
- Elle permet de sélectionner de façon précise le widget correspondant au besoin de l'utilisateur : les widgets sont classés selon les règles ergonomiques et donc en fonction d'un objectif d'utilisation.
- Elle permet d'identifier les règles s'appliquant à des groupes de widgets, via les règles ergonomiques qui sont appliquées au départ pour classer les widgets.
- Elle permet de diminuer le temps de recherche des règles et recommandations, les widgets d'une même classe étant liés par les mêmes règles et recommandations.

La classification présentée en Figure 3.13 présente quatre niveaux d'abstraction, le widget étant le premier niveau. Le second niveau d'abstraction correspond au but premier du widget :

- *Les widgets d'action* : ils permettent à l'utilisateur d'interagir avec le système pour, par exemple, entrer des données grâce à une zone d'édition de texte ou lancer un traitement au travers l'utilisation d'un bouton.
- *Les widgets d'affichage* : ils permettent au système d'afficher des informations à destination de l'utilisateur, par exemple un label permettant de caractériser un bouton.
- *Les widgets de regroupement* : ils permettent au système de regrouper d'autres widgets de manière géographique (par exemple, une boite dans laquelle sont regroupés tous les widgets pour une même fonction) ou non géographique (par exemple, une boîte sans représentation graphique permettant de mettre à jour régulièrement un groupe de widgets). Les widgets de cette classe sont les widgets containers dont nous avons parlé plus tôt.
- *Les widgets d'automatisation* : ils permettent au système de gérer le comportement autonome de l'interface. Ils peuvent par exemple envoyer des notifications à l'UA sans action de l'utilisateur sur l'interface.

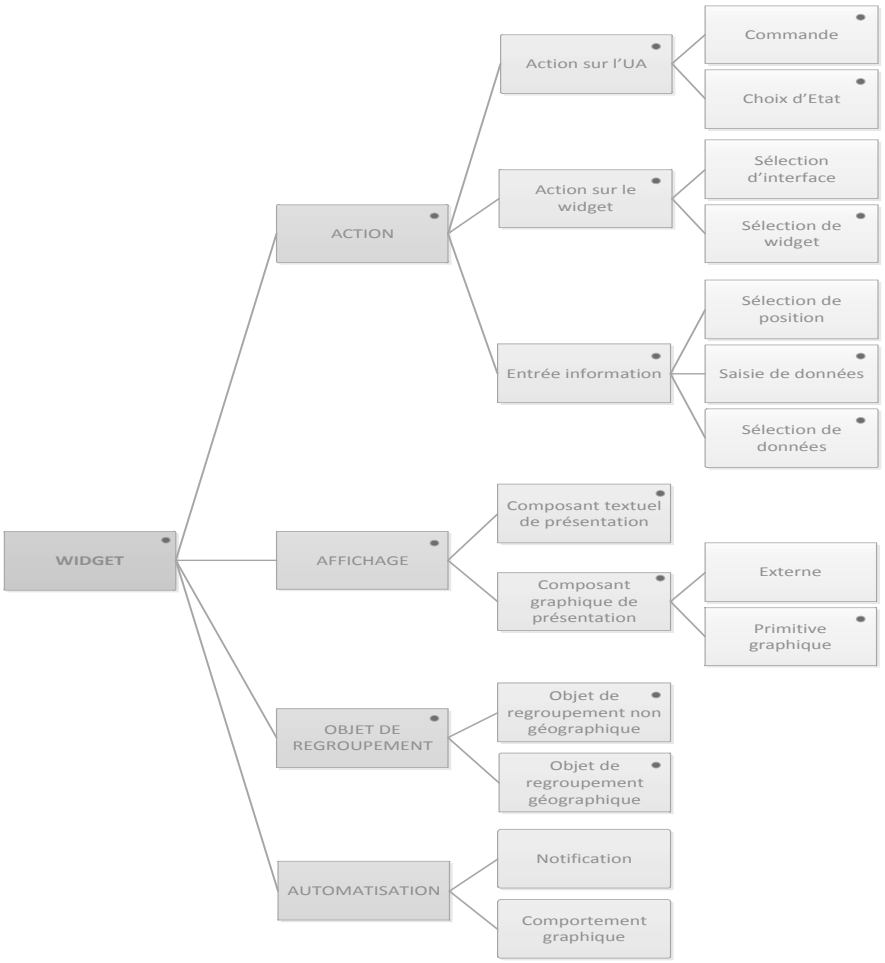


Figure 3.13. Classification des widgets du standard ARINC 661 (A. Tankeu-Choitat 2011)

3.1.2.4 Comportement du système

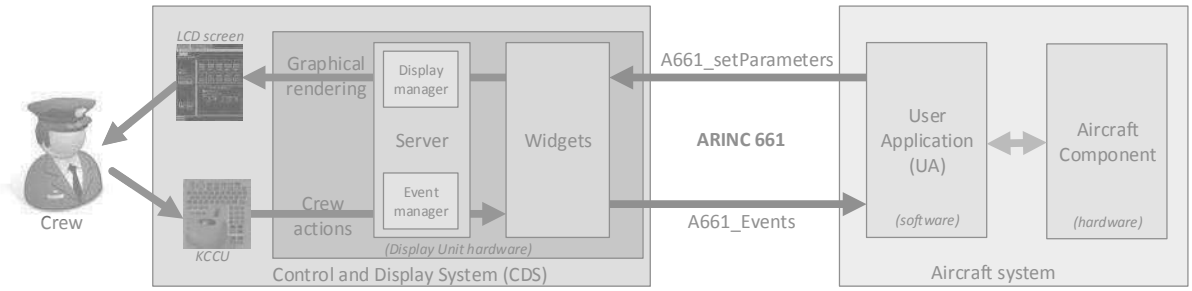


Figure 3.14. Architecture logicielle et matérielle d'un cockpit interactif respectant le standard ARINC 661

Le protocole ARINC 661 définit deux phases opérationnelles pour les applications interactives : une phase d'initialisation et une phase d'exécution. La *phase d'initialisation* correspond à l'instanciation, par le CDS, de tous les widgets définis par l'UADF ainsi que du graphe de scène associé. Pour instancier ces différents composants, le CDS utilise une bibliothèque de widgets. La *phase d'exécution* correspond à la période durant laquelle les pilotes peuvent interagir avec l'application à l'aide des KCCU. Lors de cette phase, l'architecture logicielle et matérielle du CDS correspond à celle présentée en Figure 3.14 ; afin de faciliter la compréhension du lecteur, cette architecture est beaucoup plus détaillée qu'elle ne l'est celle présentée dans le standard ARINC 661 (celle-ci ne présentant que les échanges entre le CDS et l'UA).

Durant la phase d'exécution, les échanges entre le CDS et l'UA doivent respecter le protocole ARINC 661 : le CDS notifie l'UA des actions de l'utilisateur par le biais d'événements (notés

`A661_Events`) ; l'UA demande au CDS la mise à jour des paramètres des widgets via des appels de méthodes (notées `A661_setParameters`) afin de modifier le rendu graphique de l'application pour qu'il corresponde à son état. Durant cette phase, le système effectue également un rendu graphique, communément appelé feedback. Celui-ci est local au CDS et permet aux pilotes d'avoir un retour graphique immédiat correspondant à ces actions. Nous pouvons citer comme exemple le déplacement du curseur, la mise en évidence des éléments interactifs lorsque le pilote les survole avec le curseur (appelé Hightlighting) ou encore des modifications d'affichage durant une attente de validation d'une valeur par l'UA, une animation d'enfoncement de bouton lorsque celui-ci reçoit un clic provenant de l'utilisateur…

3.2 Les contraintes de développement des systèmes avioniques critiques

Afin d'être autorisé à opérer avec des passagers, tous les avions doivent obtenir un certificat de navigabilité. Celui-ci est délivré par les autorités de navigabilité aérienne et garantit la conformité de l'avion aux normes de sécurité internationales, définies par ces mêmes autorités de navigabilité aérienne. Les principales agences de navigabilité aérienne sont l'agence européenne EASA (European Aviation Safety Agency) et l'agence américaine FAA (Federal Aviation Agency). Ces deux agences définissent et vérifient l'application des exigences concernant la conception de l'avion, son exploitation commerciale ainsi que la formation des pilotes. Ainsi, si l'on reprend l'exemple de l'Airbus A380, celui-ci a dû prouver sa compatibilité respectivement avec les normes CS 25 (EASA 2014) et 14 CFR 25 (FAA 2014) pour obtenir son certificat de navigabilité de la part respectivement de l'EASA et de la FAA.

Afin de satisfaire les exigences de certification des autorités de navigabilité aérienne, des groupes de travail ont créé des normes de développement pour l'aéronautique. Parmi ces normes, on en distingue deux particulièrement pertinentes pour le développement des systèmes avioniques critiques :

- *La norme ARP 4754* (SAE International 2010) qui propose des méthodes pour le développement des systèmes avioniques tout en prenant en compte l'ensemble de l'environnement opérationnel de l'avion et de ses fonctions. Elle définit ainsi le cycle de conception et de vie des systèmes avioniques en prenant en compte les aspects de validation des exigences des autorités de certification et de vérification de conformité des systèmes avions et de l'avion entier avec ces exigences.
- *La norme ARP 4761* (SAE International 2004) qui propose des méthodes pour évaluer la sûreté de fonctionnement des systèmes d'un avion en vue de sa certification.

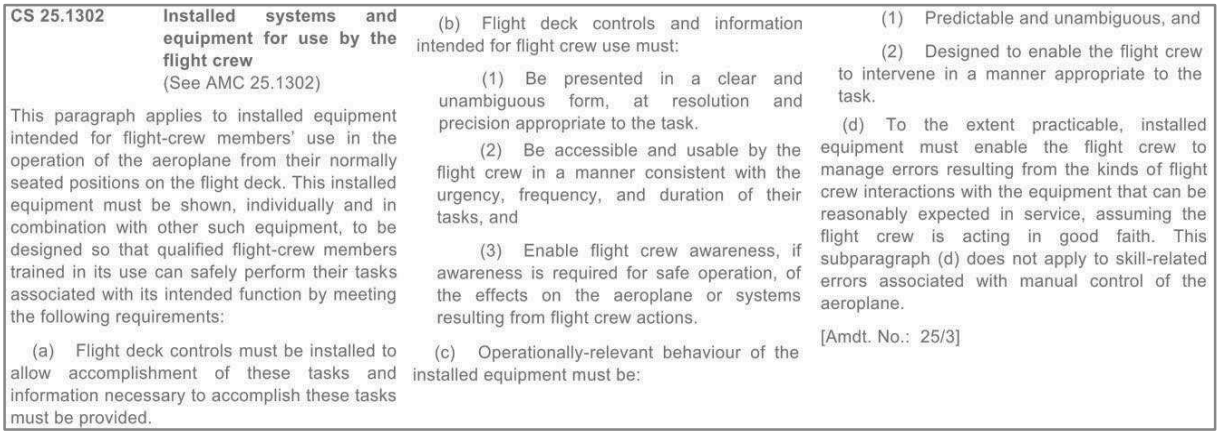
Ces normes sont complétées par d'autres normes plus précises et spécifiques sur les recommandations pour le développement des logiciels avioniques tels que la norme DO-178C (RTCA et EUROCAE 2011a) ainsi que sur les recommandations pour le développement des équipements matériels tels que la norme DO-254 (RTCA et EUROCAE 2000). Ces normes sont à appliquer par les fournisseurs de logiciel et de matériel et non par l'avionneur. Il existe également des recommandations spécifiques aux systèmes interactifs telles que l'AMC-25-11 (EASA 2014, 2.GEN.1).

Nous présentons plus en détail dans les sous-sections suivantes les normes les plus pertinentes par rapport au travail présenté dans cette thèse. Ainsi, la section 3.2.1 présente la norme *CS-25* et par conséquent les exigences de l'EASA pour la certification des systèmes interactifs. La section 3.2.2 présente la norme *DO-178C*. Pour finir, la section 3.2.3 présente les standards pour l'architecture des systèmes informatiques embarqués dans les avions, appelée *architecture modulaire avionique intégrée*.

3.2.1 La norme CS-25

La norme CS-25 (EASA 2014) décrit les spécifications que les avionneurs doivent respecter pour les avions gros porteurs (par exemple, l'Airbus A380) afin d'obtenir leur certificat de navigabilité de la part de l'EASA. Cette norme donne également des moyens de prouver la conformité aux spécifications demandées : les AMC (Acceptable Means of Compliance).

Cette thèse traitant de la sûreté de fonctionnement des systèmes interactifs, nous nous intéressons particulièrement aux parties CS-25.1302 et CS-25.1309 de la norme.



CS 25.1302 **Installed systems and equipment for use by the flight crew**
(See AMC 25.1302)

This paragraph applies to installed equipment intended for flight-crew members' use in the operation of the aeroplane from their normally seated positions on the flight deck. This installed equipment must be shown, individually and in combination with other such equipment, to be designed so that qualified flight-crew members trained in its use can safely perform their tasks associated with its intended function by meeting the following requirements:

(a) Flight deck controls must be installed to allow accomplishment of these tasks and information necessary to accomplish these tasks must be provided.

(b) Flight deck controls and information intended for flight crew use must:

(1) Be presented in a clear and unambiguous form, at resolution and precision appropriate to the task.

(2) Be accessible and usable by the flight crew in a manner consistent with the urgency, frequency, and duration of their tasks, and

(3) Enable flight crew awareness, if awareness is required for safe operation, of the effects on the aeroplane or systems resulting from flight crew actions.

(c) Operationally-relevant behaviour of the installed equipment must be:

(1) Predictable and unambiguous, and

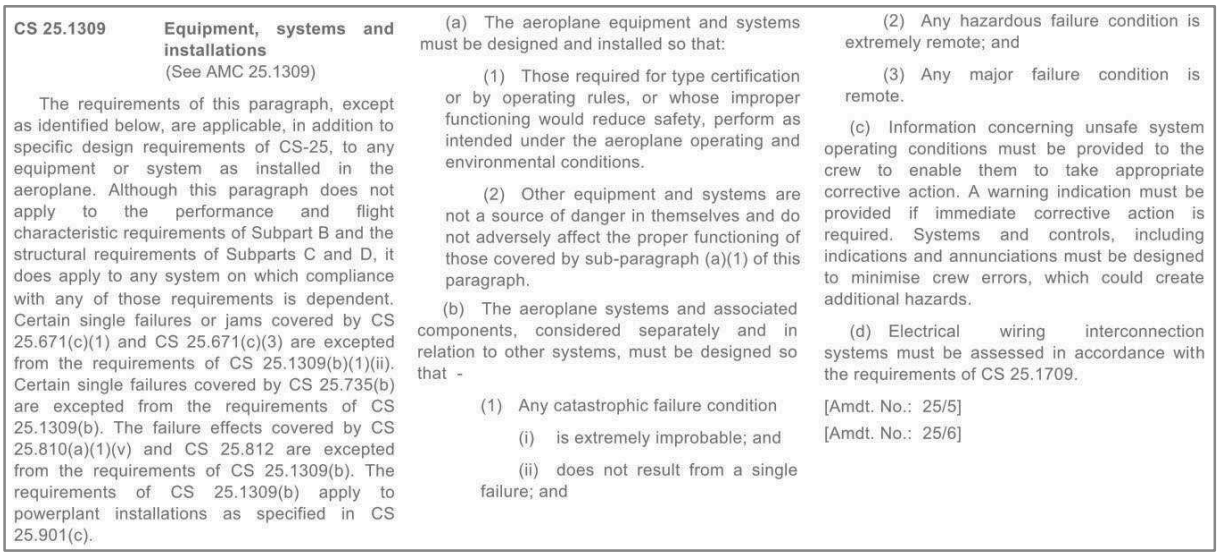
(2) Designed to enable the flight crew to intervene in a manner appropriate to the task.

(d) To the extent practicable, installed equipment must enable the flight crew to manage errors resulting from the kinds of flight crew interactions with the equipment that can be reasonably expected in service, assuming the flight crew is acting in good faith. This subparagraph (d) does not apply to skill-related errors associated with manual control of the aeroplane.

[Amdt. No.: 25/3]

Figure 3.15. CS-25.1302 (extrait de la norme CS-25 amdt 15 (EASA 2014, 1.F.1))

La CS-25.1302 (voir Figure 3.15) donne les exigences d'un point de vue de la conception des systèmes et équipements destinés à être utilisés par les pilotes pour faire voler et naviguer l'avion. Elle impose que ces systèmes de contrôle-commande doivent permettre au pilote d'accomplir toutes les tâches qu'il a à effectuer pour piloter l'avion. Les informations présentées par ces systèmes (données des systèmes avioniques et tâches accessibles aux pilotes) doivent l'être de manière claire et non ambigüe. Les tâches pilotes doivent être accessibles et utilisables d'une manière cohérente avec l'urgence, la fréquence et la durée de la tâche. Pour finir, le comportement des systèmes de commande et contrôle doit être prédictif et non ambigu et ils doivent permettre aux pilotes de gérer les erreurs pouvant survenir durant l'utilisation de tels systèmes.



CS 25.1309 **Equipment, systems and installations**
(See AMC 25.1309)

The requirements of this paragraph, except as identified below, are applicable, in addition to specific design requirements of CS-25, to any equipment or system as installed in the aeroplane. Although this paragraph does not apply to the performance and flight characteristic requirements of Subpart B and the structural requirements of Subparts C and D, it does apply to any system on which compliance with any of those requirements is dependent. Certain single failures or jams covered by CS 25.671(c)(1) and CS 25.671(c)(3) are excepted from the requirements of CS 25.1309(b)(1)(ii). Certain single failures covered by CS 25.735(b) are excepted from the requirements of CS 25.1309(b). The failure effects covered by CS 25.810(a)(1)(v) and CS 25.812 are excepted from the requirements of CS 25.1309(b). The requirements of CS 25.1309(b) apply to powerplant installations as specified in CS 25.901(c).

(a) The aeroplane equipment and systems must be designed and installed so that:

(1) Those required for type certification or by operating rules, or whose improper functioning would reduce safety, perform as intended under the aeroplane operating and environmental conditions.

(2) Other equipment and systems are not a source of danger in themselves and do not adversely affect the proper functioning of those covered by sub-paragraph (a)(1) of this paragraph.

(b) The aeroplane systems and associated components, considered separately and in relation to other systems, must be designed so that -

(1) Any catastrophic failure condition

(i) is extremely improbable; and

(ii) does not result from a single failure; and

(2) Any hazardous failure condition is extremely remote; and

(3) Any major failure condition is remote.

(c) Information concerning unsafe system operating conditions must be provided to the crew to enable them to take appropriate corrective action. A warning indication must be provided if immediate corrective action is required. Systems and controls, including indications and annunciations must be designed to minimise crew errors, which could create additional hazards.

(d) Electrical wiring interconnection systems must be assessed in accordance with the requirements of CS 25.1709.

[Amdt. No.: 25/5]

[Amdt. No.: 25/6]

Figure 3.16. CS-25.1309 (extrait de la norme CS-25 amdt 15 (EASA 2014, 1.F.3))

La CS-25.1309 (voir Figure 3.16) donne les exigences de sûreté de fonctionnement pour tous les systèmes avioniques. Elle impose de garantir le fonctionnement correct en vol de tous les systèmes avioniques critiques, c'est-à-dire les systèmes dont le dysfonctionnement diminuerait la sécurité-innocuité (`safety`) de l'avion. De la même manière, cette norme impose de garantir que les autres systèmes avioniques ne puissent pas affecter le comportement des systèmes critiques. Cette norme donne également des objectifs de probabilité d'occurrence des défaillances en fonction de leur sévérité :

- Toute défaillance *catastrophique* doit être extrêmement improbable et ne doit pas découler de la défaillance d'un seul système (ce dernier critère est appelé *fail-safe*) ;
- Toute défaillance *dangereuse* doit être très improbable ;

- Toute défaillance *majeure* doit être improbable.

Ces objectifs sont définis dans l'AMC-25.1309 (EASA 2014, 2.F.38) qui fournit des moyens de prouver la conformité avec la CS-25.1309.

a. *Classifications.* Failure Conditions may be classified according to the severity of their effects as follows:

(1) *No Safety Effect:* Failure Conditions that would have no effect on safety; for example, Failure Conditions that would not affect the operational capability of the aeroplane or increase crew workload.

(2) *Minor:* Failure Conditions which would not significantly reduce aeroplane safety, and which involve crew actions that are well within their capabilities. Minor Failure Conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some physical discomfort to passengers or cabin crew.

(3) *Major:* Failure Conditions which would reduce the capability of the aeroplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to the flight crew, or physical distress to passengers or cabin crew, possibly including injuries.

(4) *Hazardous:* Failure Conditions, which would reduce the capability of the aeroplane or the ability of the crew to cope with adverse operating, conditions to the extent that there would be:

(i) A large reduction in safety margins or functional capabilities;

(ii) Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely; or

(iii) Serious or fatal injury to a relatively small number of the occupants other than the flight crew.

(5) *Catastrophic:* Failure Conditions, which would result in multiple fatalities, usually with the loss of the aeroplane. (Note: A "Catastrophic" Failure Condition was defined in previous versions of the rule and the advisory material as a Failure Condition which would prevent continued safe flight and landing.)

Figure 3.17. Classification des défaillances (extrait de la norme CS-25 (EASA 2014, 2.F.43))

Ainsi, l'AMC-25.1309 propose une classification des modes de défaillances avec une échelle de criticité à cinq niveaux en fonction de leur impact sur l'avion (voir Figure 3.17) :

- *Catastrophique (`Catastrophic`) :* la défaillance peut amener à de nombreuses pertes humaines (en provoquant, par exemple, le crash de l'avion) ;
- *Dangereux (`Hazardous`) :* la défaillance crée un lourd impact négatif sur le fonctionnement de l'avion ou son contrôle par les pilotes et peut amener à des blessures importantes chez les passagers ;
- *Majeur (`Major`) :* la défaillance réduit de manière significative la sécurité-innocuité de l'avion ou augmente de manière conséquente le travail des pilotes, elle peut amener à des pertes de confort pour les passagers, pouvant aller jusqu'à des blessures bénignes ;
- *Mineur (`Minor`) :* réduit quelque peu la sécurité-innocuité de l'avion ou augmente un peu le travail des pilotes, elle peut amener à des gênes physiques pour les passagers (telles que des turbulences) ;
- *Sans effet sur la sécurité-innocuité (`No Safety Effect`) :* la défaillance n'a aucun effet sur la sécurité-innocuité de l'avion.

L'AMC-25.1309 définit également les objectifs de probabilité d'occurrence en fonction de la sévérité de la défaillance. Ces objectifs sont résumés dans le Tableau 3.1.

Severity of failure conditions	Probability of failure condition	Quantitative probability of failure condition (per flight hour)
Catastrophic	Extremely improbable	$< 10^{-9}$
Hazardous	Extremely remote	$< 10^{-7}$
Major	Remote	$< 10^{-5}$
Minor	Probable	$< 10^{-3}$
No Safety Effect	No probability requirement	No probability requirement

Tableau 3.1. Relation entre la sévérité et la probabilité d'occurrence des défaillances

La norme CS-25 propose également toute une partie regroupant des recommandations spécifiques aux systèmes interactifs : l'AMC 25-11 (EASA 2014, 2.GEN.1).

L'AMC-25-11 donne notamment des tâches et moyens pour prouver la conformité des systèmes interactifs à la CS-25.1302 en donnant des recommandations sur la présentation de l'information, l'utilisation des couleurs, la gestion de l'information et les moyens d'interaction avec l'équipage. Ces recommandations sont à compléter avec l'AMC-25.1302 (EASA 2014, 2.F.1).

Pour satisfaire les exigences de la CS 25.1309, l'AMC 25-11 propose trois moyens principaux :

- L'identification des modes de défaillance du système interactif. Les défaillances possibles concernent la perte et les dysfonctionnements en valeur (appelés erronées) des fonctions d'affichage et de contrôle.
- La description des effets de ces défaillances sur le pilotage de l'avion, la charge de travail des pilotes. Il est important de noter que les effets des défaillances des systèmes de commande et contrôle sont dépendants de nombreux facteurs tels que les compétences du pilote, la phase de vol…
- La proposition de moyens d'atténuation des défaillances. Dans ce cas, il s'agit de décrire des mécanismes de sûreté de fonctionnement permettant de traiter ou réduire les effets des défaillances et leur sévérité.

Enfin, il est important de rappeler que l'AMC 25-11 traite des systèmes interactifs de l'avion, qui ont pour vocation d'être utilisés par les pilotes de l'avion. Il est donc important de ne pas négliger le composant humain du système interactif. Pour cela, l'AMC 25-11 recommande de documenter les aspects performances de l'utilisateur lors du développement du système. Il s'agit donc de documenter :

- La charge de travail des pilotes, que l'avion soit en fonctionnement standard ou en mode de fonctionnement dégradé (par exemple, lors de la réception d'une alarme) ;
- Le temps de formation nécessaire aux pilotes pour se familiariser avec le système ;
- Les risques potentiels d'erreur des pilotes.

3.2.2 La norme DO-178C

La norme DO-178C (RTCA et EUROCAE 2011a) est un standard pour le développement logiciel dans le domaine de l'aéronautique. Son but est de donner des directions pour la production de logiciel pour les équipements avions. Pour cela, elle propose des objectifs à atteindre tout au long du cycle de vie du logiciel ainsi que des tâches pour les atteindre et des moyens pour prouver qu'ils ont bien été atteints.

Plus précisément, la norme DO-178C s'appuie à la fois sur le processus de développement des systèmes de la norme ARP 4754 (SAE International 2010) et sur l'analyse de sûreté de fonctionnement des systèmes de la norme ARP 4761 (SAE International 2004) pour définir des processus de développement pour le logiciel couvrant tout son cycle de vie :

- *Le processus de planification* qui définit et coordonne les activités des processus de développement et d'intégration pour un logiciel ;
- *Le processus de développement* qui permet la création du logiciel, il comprend 4 étapes
    - La phase d'analyse des exigences (`SW Requirement Process`) qui produit les exigences de haut niveau (incluant les exigences fonctionnelles et non-fonctionnelles telles que les exigences de sûreté de fonctionnement) ;
    - La phase de design (`SW Design Process`) qui permet de raffiner les exigences de haut niveau en une architecture logicielle et des exigences bas niveau qui pourront être utilisées directement pour le codage ;
    - La phase de codage (`SW Coding Process`) qui permet la mise en œuvre du code source à partir de l'architecture logicielle et des exigences de bas niveau ;

- La phase d'intégration (`Integration Process`) qui permet de créer le logiciel exécutable et de le lier au système intégré.

- *Les processus intégraux* qui permettent d'assurer la conformité du logiciel. Ils comprennent la vérification, la validation, la gestion de configuration, l'assurance qualité et la coordination pour la certification.

La Figure 3.18 présente l'intégration des processus de développement logiciel définis par la norme DO-178C avec les processus de développement et d'analyses de la norme ARP 4754 et de la norme ARP 4761.

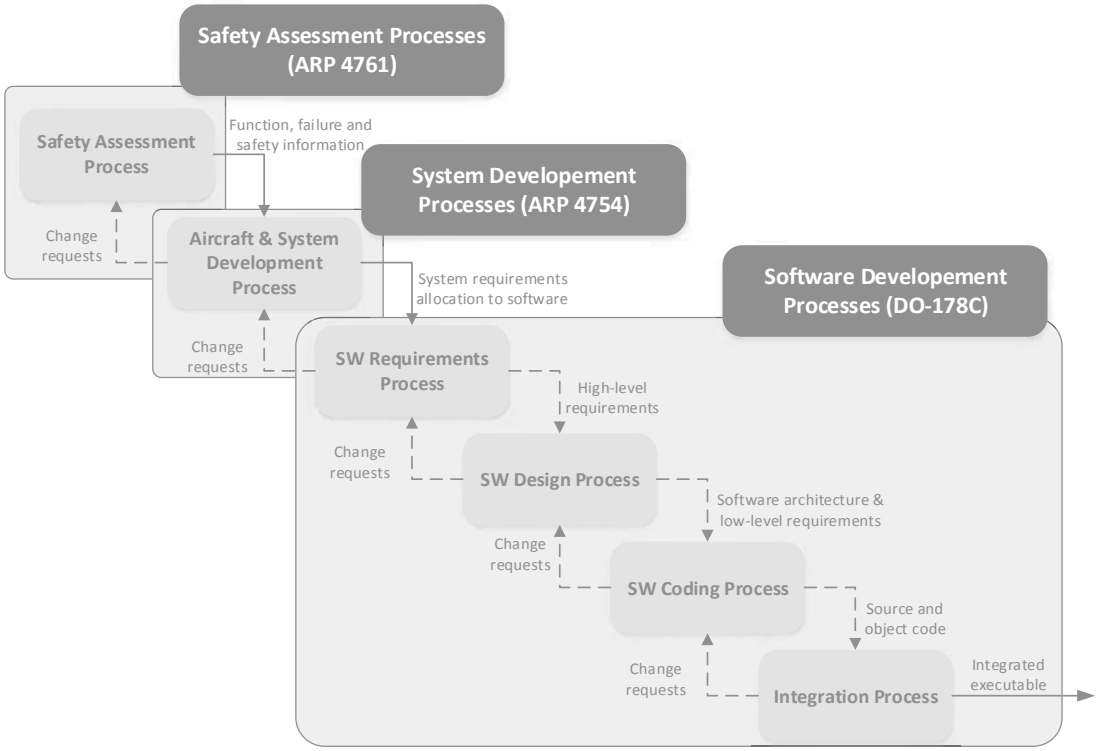


Figure 3.18. Processus de développement logiciel de la norme DO-178C

En plus des directions proposées pour le processus de développement logiciel, la norme DO-178C met l'accent sur les aspects sécurité-innocuité (`safety`) du logiciel afin de pouvoir garantir sa sûreté de fonctionnement. La démarche proposée repose sur un processus d'analyse de la sécurité-innocuité des systèmes comme celui proposé par la norme ARP 4761. Ce processus, qui repose sur des études de risques préliminaires (autant fonctionnelles qu'architecturales), consiste à analyser les fonctions et l'architecture des systèmes avioniques de manière à identifier leurs modes de défaillance et leurs potentiels effets sur l'avion.

Pour pouvoir garantir la sûreté de fonctionnement du logiciel en fonction de son niveau de criticité, la norme DO-178C définit cinq niveaux d'assurance classés sur une échelle appelée DAL (Development Assurance Level), allant de A à E. Cette échelle s'aligne sur les niveaux de sévérité des défaillances définis par la norme AMC-25.1309. Ainsi, le niveau DAL A correspondant au niveau de défaillance catastrophique et le niveau DAL E au niveau sans effet sur la sécurité-innocuité. Le Tableau 3.2 résume les correspondances entre les niveaux de criticité des modes de défaillance de l'AMC-25.1309, les niveaux de DAL définis par la norme DO-178C et les objectifs quantitatifs en termes de probabilité de défaillance par heure de vol de l'avion.

Il est important de noter que les systèmes pouvant provoquer des défaillances catastrophiques nécessitent, en plus du niveau DAL A, la couverture du critère *fail-safe*. Ce critère impose qu'aucune panne simple ne puisse conduire à une défaillance catastrophique. Pour atteindre cet objectif, les systèmes avioniques sont généralement développés en ayant recours à la redondance et à la tolérance aux fautes.

Severity of failure conditions	DAL level	Quantitative probability of failure condition (per flight hour)
Catastrophic	A + Fail Safe	$< 10^{-9}$
Hazardous	B	$< 10^{-7}$
Major	C	$< 10^{-5}$
Minor	D	$< 10^{-3}$
No Safety Effect	E	No probability requirement

Tableau 3.2. Les niveaux de criticité, DAL et objectifs quantitatifs

Bien que la norme DO-178C ne l'impose pas, elle recommande fortement d'utiliser des stratégies de prévention des fautes lors du développement. Ainsi, elle recommande par exemple l'utilisation de méthodes formelles qui sont des techniques de spécification, développement et vérification du logiciel fondées sur des preuves mathématiques. L'utilisation de ces méthodes peut participer aux efforts de fiabilisation du logiciel car elles permettent de décrire de manière non ambigüe les exigences du logiciel ainsi que la vérification de l'exactitude et la cohérence du logiciel par rapport à ces spécifications. Les méthodes formelles peuvent être appuyées par un développement par les modèles du logiciel. Les recommandations pour l'utilisation des méthodes formelles dans le cycle du développement logiciel ont été standardisées dans un supplément à la norme DO-178C : la norme DO-333 (RTCA et EUROCAE 2011c). De la même manière, les recommandations pour l'intégration du développement par les modèles dans le cycle de vie du développement logiciel ont été standardisées dans un supplément à la norme DO-178C : la norme DO-331 (RTCA et EUROCAE 2011b).

3.2.3 Les standards pour l'architecture avionique modulaire intégrée

Depuis les années 90, l'informatique embarquée dans les avions repose sur une architecture modulaire appelée *avionique modulaire intégrée* (ou IMA pour Integrated Modular Avionics) (Prisaznuk 1992). Cette architecture repose sur le partage des ressources de calcul et de communications entre les fonctions et est définie par le standard ARINC 651 (AEEC 1991). Un des principaux avantages d'une telle architecture est qu'elle permet de faciliter la redondance des composants informatiques ainsi que leur reconfiguration en cas de défaillance. Elle permet également de diminuer le poids, la consommation de l'avion et les coûts de développement et de maintenance de l'informatique embarquée. L'architecture du CDS présentée en section 3.1.1 respecte cette architecture.

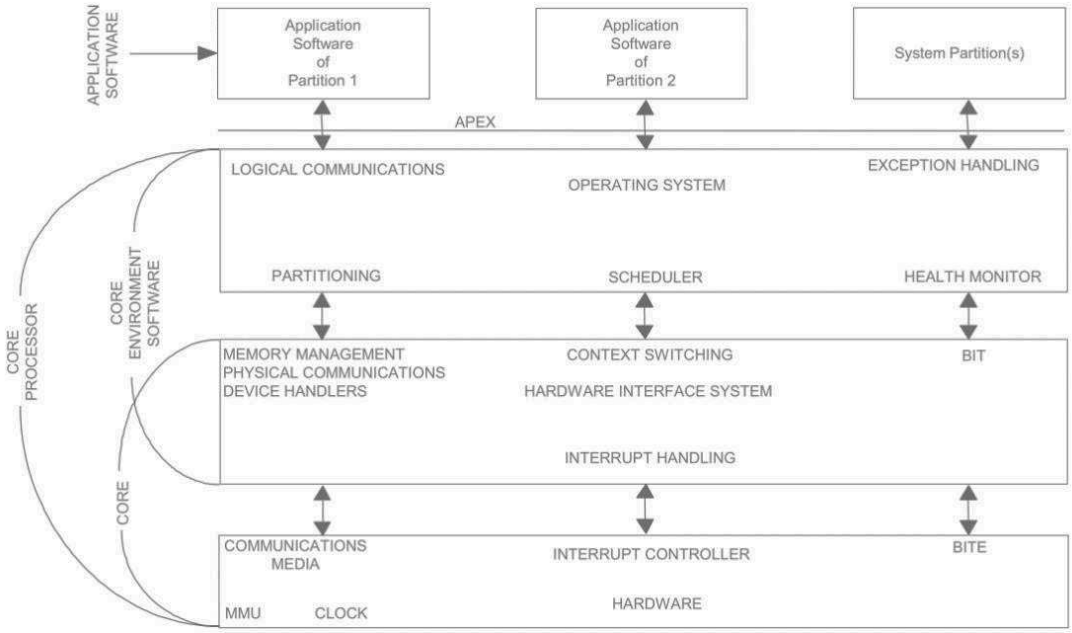


Figure 3.19. Architecture définie par le standard ARINC 653 (AEEC 2003)

Le standard ARINC specification 653 (AEEC 2003) définit le comportement et l'interface des systèmes d'exploitation temps réel embarqués dans l'avionique. Il définit une interface qui permet le développement en parallèle de plusieurs composants par des constructeurs différents, tout en garantissant un support commun pour ces composants (Prisaznuk 2008). Le principal avantage de ce standard est qu'il définit des partitions (voir Figure 3.19) assurant l'exécution de plusieurs applications sur une même plateforme matérielle. Une partition est une zone de mémoire protégée à qui a été affectée une tranche de temps, répétée périodiquement, pendant laquelle elle est la seule à pouvoir s'exécuter sur la plateforme. Le standard ARINC 653 permet, avec cette définition des partitions, l'application aux systèmes avioniques des concepts de ségrégation spatiale et temporelle, également appelés TSP (Time and Space Partioning) (Rushby 1999).

La ségrégation spatiale impose le confinement de la mémoire d'une application. Ainsi, une partition ARINC 653 ne peut pas modifier la mémoire d'une autre partition. La ségrégation temporelle impose qu'à chaque partition est affectée une tranche de temps unique, ce qui permet d'imposer un partage des ressources déterministe entre les différentes partitions. L'application de ces deux principes garantit le confinement des partitions. C'est-à-dire qu'aucune propagation n'est possible d'une partition à l'autre.

De plus, le standard ARINC 653 définit également des moyens de communication entre les différentes partitions. On distingue ainsi deux moyens de communication différents :

- *Le queuing* : il permet la communication par queue FIFO (First-In First-Out). Concrètement, il permet l'envoi de plusieurs messages en file d'attente par la partition émettrice de manière à ce que le premier message envoyé soit le premier message lu. La lecture d'un message par la partition réceptrice est effaçante : elle libère l'espace mémoire de ce message dans la file d'attente qui peut alors recevoir un nouveau message. Aucun message ne peut être effacé avant d'avoir été lu par la partition réceptrice.
- *Le sampling* : il permet la communication par tableau. Concrètement, la communication comprend un unique espace mémoire qui peut être vu comme un tableau noir : lorsque la partition émettrice envoie un message, celui-ci efface ce qui pouvait être contenu dans le tableau précédemment. La lecture par la partition réceptrice n'est pas effaçante.

Ainsi, la communication par *queuing* est particulièrement adaptée pour l'envoi de messages multiples dans le cas où ils sont tous importants (par exemple, des commandes à effectuer par un système). La communication par *sampling* quant à elle est particulièrement adaptée pour l'envoi de messages où seule la dernière valeur est importante (par exemple, la mise à jour de la valeur d'un capteur).

Pour conclure, le standard ARINC 653 est particulièrement intéressant pour l'approche présentée dans cette thèse car il définit les principes de ségrégation spatiale et temporelle pour les systèmes informatisés dans l'avionique. Il permet donc la définition de zones de confinement entre lesquelles la propagation de fautes n'est pas possible, les rendant ainsi particulièrement adaptées aux principes de tolérance aux fautes présentés en section 2.6.

3.3 Exigences de sûreté de fonctionnement

Les systèmes de commande et contrôle dans le cockpit ont beaucoup évolués ces dernières années. Ainsi, les cockpits ont pu passer des systèmes de commande et contrôle analogiques où les commandes étaient effectuées à l'aide de boutons poussoirs et rotatifs et les affichages à l'aide de jauges mécaniques et d'instruments électromagnétiques, aux glass cockpits où les informations provenant de plusieurs systèmes avioniques peuvent être regroupées sur un même écran mais où les commandes se font grâce à des boutons physiques sans intégration des entrées et sorties. La dernière génération de cockpits (les cockpits interactifs) intègre un CDS où les pilotes peuvent interagir directement sur un certain nombre d'applications interactives respectant le standard ARINC 661 via un dispositif de clavier et souris.

Les exigences de sûreté de fonctionnement pour la certification des avions imposent que les systèmes de commande et contrôle soient construits avec le même niveau de DAL que les systèmes qu'ils permettent de commander. Ainsi, les systèmes critiques, nécessitant un niveau d'assurance DAL A ou DAL B, doivent être commandés et contrôlés par des systèmes interactifs développés au niveau

DAL A ou DAL B. Les systèmes de commande et contrôle historiques (analogiques et glass cockpits) sont depuis longtemps développés en intégrant des mécanismes de tolérance aux fautes tels que le COM/MON (Traverse, Lacaze et Souyris 2004) qui permettent d'atteindre le niveau d'assurance souhaité (DAL A ou DAL B) ainsi que le critère fail-safe.

À l'heure actuelle, les systèmes interactifs ne sont utilisés que pour la commande et le contrôle de systèmes avioniques non critiques, c'est-à-dire, des systèmes dont la défaillance n'aura pas d'impact supérieur à *Majeur* selon la classification de l'ARP 4764 et la DO-178C. Ces systèmes interactifs sont donc développés à l'heure actuelle à un niveau de développement maximal DAL C. Il est important de noter que ces restrictions ne concernent que la partie interactive du cockpit.

Ces restrictions sont mises en évidence par la Figure 3.20 où nous pouvons voir que certaines zones du CDS sont non-interactives, l'utilisation du curseur y est impossible et interdite. En effet, les applications affichées sur ces zones sont uniquement des applications d'affichage d'information. Elles sont développées à la manière des applications chargées de l'affichage dans les glass cockpits et elles respectent donc les niveaux d'assurance DAL A ou DAL B. Les zones encadrées en vert sur la Figure 3.20 représentent les zones où s'appliquent les applications interactives respectant le standard ARINC 661.

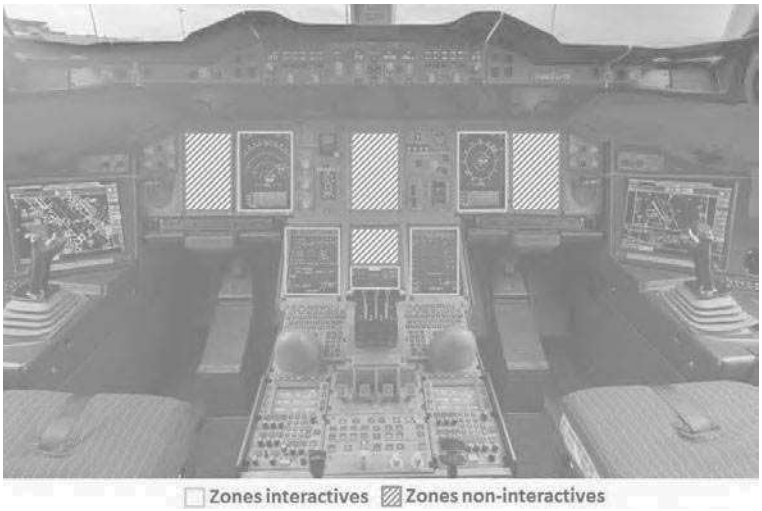


Figure 3.20 Zones interactives dans le CDS de l'Airbus A380

L'industrie avionique s'intéresse actuellement à la possibilité de commander et contrôler des systèmes avions critiques grâce à l'utilisation des systèmes interactifs modernes (cockpits interactifs). Pour cela il est nécessaire que le développement de ces systèmes réponde aux exigences de sûreté de fonctionnement de plus haut niveau des autorités de certification. Il est nécessaire donc être capable de garantir un niveau d'assurance de développement équivalent au niveau DAL A respectant le critère fail-safe pour ces systèmes interactifs. Il s'agit donc d'appliquer les méthodes et outils conseillés pour les systèmes avioniques critiques à ces systèmes interactifs : mettre en place des processus de développement de système critique permettant de s'assurer du fait d'avoir un logiciel sans défaut et d'intégrer des mécanismes de tolérance aux fautes pour permettre d'atteindre les faibles probabilités de défaillances souhaitées.

3.4 Synthèse

Ce chapitre a permis de mettre en évidence les systèmes ciblés par la thèse : les systèmes interactifs des cockpits d'avions civils. Ceux-ci sont fortement contraints par des exigences liées à la certification de l'avion.

Leur architecture logicielle doit respecter le standard ARINC specification 661. Celui-ci définit à la fois l'interface logicielle des objets interactifs qui pourront être utilisés par les pilotes (les widgets), le protocole de communication entre ces widgets et la partie logicielle des systèmes avioniques qu'ils permettent de contrôler et le comportement des systèmes interactifs dans le cockpit.

Leur développement doit respecter les exigences de l'EASA définies dans la norme CS-25 pour pouvoir être certifiés. Pour cela, il est nécessaire de suivre les processus de développement définis par les normes ARP 4754 et ARP 4761 ainsi que la norme DO-178C.

L'architecture des systèmes informatiques embarqués dans l'avion est définie par les standards ARINC 651 et 653. Le standard ARINC 653 est particulièrement intéressant pour l'approche présentée dans cette thèse car il définit les principes de ségrégation spatiale et temporelle pour les systèmes informatisés dans l'avionique. Il permet la définition de zones de confinement entre lesquelles la propagation de fautes n'est pas possible, les rendant ainsi particulièrement adaptées aux principes de tolérance aux fautes

Les systèmes interactifs doivent être développés de manière à atteindre un niveau d'assurance équivalent à celui des systèmes dont ils permettent d'assurer le contrôle. Les systèmes interactifs sont développés à l'heure actuelle avec un niveau d'assurance DAL C, ils ne servent donc qu'au contrôle des systèmes non-critiques. Les systèmes critiques sont contrôlés grâce à des systèmes interactifs « historiques » (analogiques et glass cockpits) qui respectent des niveaux d'assurance DAL A et DAL B. Pour pouvoir commander et contrôler les systèmes critiques de l'avion à l'aide des systèmes interactifs, il est nécessaire de mettre en place des méthodes et moyens pour permettre d'assurer, à travers leur développement, un niveau d'assurance maximal : le niveau DAL A avec le respect du critère *fail-safe*.

Ce chapitre nous a donc permis de mettre en évidence les contraintes dont nous devrons tenir compte tout au long de cette thèse. Ainsi, pour concevoir et développer des systèmes interactifs sûrs de fonctionnement et utilisables, il est nécessaire :

- Que ceux-ci respectent le standard ARINC 661 ;
- Que le support sur lequel ils sont exécutés respecte les standards ARINC 651 et 653
- Que leur développement respecte les normes ARP 4754, ARP 4761 et DO-178C
- Qu'ils soient développés pour respecter un niveau d'assurance maximal : le niveau DAL A avec le respect du critère *fail-safe*.

Partie 2. CONTRIBUTIONS

Contenu de la partie

Cette deuxième partie présente l'approche globale que nous proposons pour concevoir et développer des systèmes interactifs sûrs de fonctionnement. Celle-ci est composée d'une approche à base de modèles visant à réaliser des systèmes interactifs zéro-défaut et d'une architecture logicielle générique pour des systèmes interactifs tolérants aux fautes.

Le Chapitre 4 définit le périmètre des travaux que nous présentons en détaillant les hypothèses que nous avons faites, les modes de défaillances que nous cherchons à éviter ainsi que le modèle de faute regroupant les fautes que nous pouvons traiter grâce à l'utilisation de notre approche. Nous définissons ensuite une architecture générique, fondée sur le modèle architectural ARCH pour les systèmes interactifs afin d'en détailler tous les composants logiciels. Pour finir, nous présentons les principes généraux de l'approche globale.

Le Chapitre 5 définit une approche à base de modèles pour la conception et le développement de systèmes interactifs zéro-défaut. Plus concrètement, nous proposons de modéliser tous les composants logiciels des systèmes interactifs à l'aide d'une notation formelle nommée ICO.

Le Chapitre 6 définit une architecture logicielle générique et tolérante aux fautes pour les systèmes interactifs. Cette architecture est fondée sur l'application du principe de la programmation n-autotestable à tous les composants logiciels des systèmes interactifs.

Le Chapitre 7 présente des moyens pour la mise en œuvre de notre approche pour des systèmes sûrs de fonctionnement. Nous proposons premièrement l'utilisation d'un outil pour la modélisation des composants logiciels avec la notation formelle ICO. Deuxièmement, nous proposons une mise en œuvre pour notre architecture tolérante aux fautes, fondée sur l'utilisation de l'outil pour la modélisation et d'un simulateur de système d'exploitation avionique. Enfin, nous proposons des pistes pour la validation de notre approche.

Sommaire

Chapitre 4. Définition du périmètre et des principes de l'approche

Sommaire

Ce chapitre définit les principes de l'approche que nous proposons pour concevoir et développer des systèmes interactifs sûrs de fonctionnement. Pour cela, il est tout d'abord nécessaire de définir l'objectif concret de cette approche ainsi que son périmètre et les hypothèses qui définissent l'ensemble des éléments traités et ceux qui sont considérés au-delà de ce travail. Il est également nécessaire de définir les modes de défaillance que nous cherchons à éviter avec l'application de notre approche et le modèle de fautes que nous considérons dans cette thèse. Enfin, il est nécessaire de définir une représentation formelle (sous forme d'architecture) d'un système interactif afin de définir les éléments logiciels d'un système interactif sur lesquels nous pourrons appliquer notre approche. Nous proposons pour cela une architecture générique pour les systèmes interactifs dont les composants pourront être utilisés pour la mise en œuvre de notre approche. Ayant défini tous ces prérequis, nous pouvons alors présenter notre approche fondée sur deux méthodes, la première permettant la conception zéro-défaut des systèmes interactifs et la seconde permettant la conception de systèmes interactifs tolérants aux fautes.

La première section définit le périmètre couvert par cette thèse et les hypothèses que nous faisons pour définir les éléments traités et ceux se trouvant en dehors du périmètre.

La deuxième section présente les modes de défaillance que nous cherchons à éviter et le modèle de fautes considéré.

La troisième section présente une architecture générique pour les systèmes interactifs sur laquelle nous nous appuyons pour appliquer notre approche.

Enfin, la quatrième section présente notre approche pour concevoir et développer des systèmes interactifs sûrs de fonctionnement.

4.1 Périmètre et hypothèses

4.1.1 Périmètre

L'objectif de cette thèse est de proposer des méthodes et outils permettant de spécifier et développer des systèmes interactifs sûrs de fonctionnement. Ils permettent de garantir le bon fonctionnement en opération de tels systèmes et donnent des moyens pour les développer avec des niveaux de fiabilité équivalents à ceux des systèmes embarqués critiques tels que les commandes de vols.

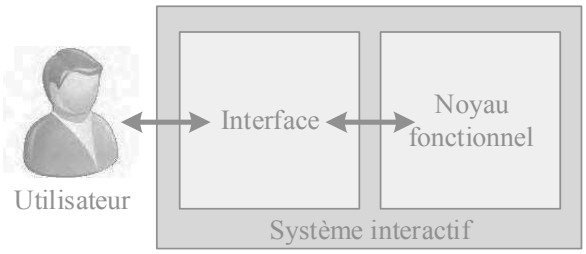


Figure 4.1. Modèle simplifié d'un système interactif

Nous nous intéressons à la partie informatique des systèmes interactifs et plus particulièrement aux composants informatiques responsables des fonctionnalités interactives. Ces composants correspondent à la partie interface du modèle simplifié des systèmes interactifs rappelé en Figure 4.1 ou encore aux parties contrôleur de dialogue, interaction logique et interaction physique du modèle architectural ARCH rappelé en Figure 4.2 (voir section 2.4.1). Les erreurs humaines sont donc hors du périmètre de notre étude.

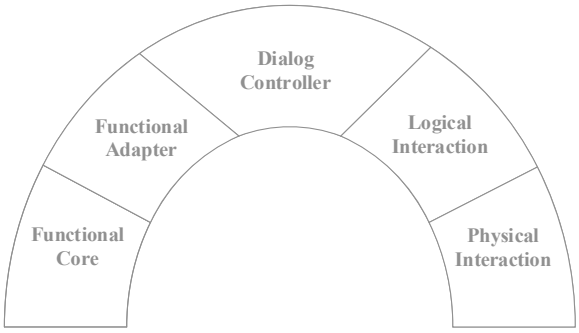


Figure 4.2. Modèle ARCH d'un système interactif

Cependant, la sûreté de fonctionnement et l'utilisabilité sont les propriétés non fonctionnelles les plus importantes pour les systèmes que nous considérons (voir section 2.2.3). Ces deux propriétés étant orthogonales, il est important de s'assurer que l'approche proposée n'impacte pas l'utilisabilité du système. En effet, une diminution de l'utilisabilité du système pourrait créer une augmentation de la charge de travail des pilotes en opération. Nous avons travaillé sur ces aspects. Bien que ce travail ne soit pas assez abouti pour figurer comme une contribution à part entière de la thèse, nous proposons dans la section 7.3.3 des pistes pour s'assurer que l'approche proposée n'implique pas une diminution de l'utilisabilité du système.

Enfin, nous nous intéressons plus particulièrement aux techniques d'interactions WIMP (van Dam 1997) pour lesquelles l'affichage et la manipulation se font au travers d'un ensemble prédéfini de widgets (voir section 2.1.1). Ces techniques d'interaction ont été étudiées et standardisées par IBM en 1989 (IBM 1989). Elles sont notamment utilisées dans les systèmes interactifs critiques comme dans l'avionique où elles ont été standardisées par le standard ARINC 661 (AEEC 2013) qui définit les interactions dans les cockpits d'avions civils (voir section 3.1.2.1).

4.1.2 Hypothèses

Dans un premier temps, il est important de noter que nous nous intéressons aux systèmes interactifs en tant que systèmes informatiques. Les erreurs potentielles pouvant provenir de l'utilisateur sont donc hors du périmètre de nos travaux. Premièrement, nous ne considérons pas les erreurs humaines, nous ne cherchons pas à les tolérer ou à diminuer le nombre de leurs occurrences. Nous ne nous intéressons donc pas aux aspects du système pouvant être source d'erreur humaine tels que son utilisabilité ou son design. Nous sommes conscients que cette hypothèse est très forte car les utilisateurs font des erreurs mais l'approche que nous proposons dans ces travaux ne permet pas de couvrir ces erreurs. Deuxièmement, nous ne considérons pas non plus les attaques malveillantes pour lesquelles l'utilisateur chercherait à nuire au système. Celles-ci relèvent en effet d'un objectif de sécurité (`security`) et non de celui de sûreté de fonctionnement (`dependability`) que nous nous sommes fixés. Cette hypothèse est justifiée car les systèmes interactifs critiques (et donc, ceux qui nécessitent de la sûreté de fonctionnement) sont généralement confinés, isolés de l'extérieur, ce qui permet d'assurer le fait qu'une personne mal intentionnée ne pourra pas le modifier.

Dans un second temps, le périmètre des travaux de cette thèse se concentre plus particulièrement sur les composants logiciels responsables des fonctionnalités interactives du système (les composants de l'interface), ce qui nous amène à faire les hypothèses suivantes :

H1 : Le noyau fonctionnel ainsi que les données qu'il transmet à la partie interface du système sont sûrs de fonctionnement.

Le *noyau fonctionnel* correspond à la partie du système interactif contenant les fonctionnalités non-interactives du système, il est très dépendant de son domaine d'application. Nous le considérons hors du champ de cette étude. Cette hypothèse est d'autant plus justifiée lorsque l'on considère les systèmes interactifs des cockpits interactifs. En effet, dans ce cas, le noyau fonctionnel est composé d'un système avionique (par exemple, les commandes de vol électriques). La sûreté de fonctionnement d'un tel système est assurée par ailleurs. En effet, ces systèmes intègrent des principes de ségrégation, redondance et diversification (par exemple les commandes de vol électriques de la famille des Airbus A320 (Traverse, Lacaze et Souyris 2004) ou encore celles des Boeing B777 (Y. C. Yeh 1998)).

H2 : La communication entre l'interface du système interactif et son noyau fonctionnel est sûre de fonctionnement : le transfert de données se fait sans corruption.

Cette hypothèse est tout à fait justifiée car il existe de nombreux protocoles de communication fiables tels que le réseau AFDX (AEEC 2009) qui est présent dans les avions modernes d'Airbus (par exemple l'A380) et qui spécifie des mécanismes de tolérance aux fautes (Wang, Wang et Shi 2012).

H3 : Les périphériques d'entrée et de sortie du système interactifs et leurs pilotes informatiques (drivers) sont sûrs de fonctionnement.

Nous considérerons que les données envoyées par les périphériques d'entrée (par exemple un clavier et une souris), via leurs drivers sont fiables, c'est-à-dire qu'elles ne sont pas corrompues. Nous considérons également que les données envoyées aux périphériques de sortie (par exemple un écran) sont affichées de manière correcte et complète. De plus il existe à l'actuelle des moyens en place pour les rendre sûrs de fonctionnement tels que la duplication des capteurs de pressions sous un bouton comme c'est le cas à l'heure actuelle pour les boutons analogiques.

H4 : Les composants matériels utilisés pour l'exécution du logiciel du système interactif sont sûrs de fonctionnement.

Nous considérons les fautes affectant le matériel lors de son développement en dehors du périmètre de cette étude. En effet, les fautes affectant les composants matériels durant le développement du système doivent être adressées par les fabricants de matériel et très peu de choses peuvent être faites pour y remédier au niveau du développement du système (mises à part la redondance et la diversification du support d'exécution en utilisant, par exemple, deux processeurs différents pour la même fonction).

4.2 Modes de défaillance et modèle de faute pour les systèmes interactifs

Pour garantir le fonctionnement correct d'un système interactif, il faut tout d'abord identifier les entraves à la sûreté de fonctionnement qui peuvent l'affecter. La chaîne de causalité des entraves à la sûreté de fonctionnement est rappelée en Figure 4.3. Dans cette optique, nous déterminons dans cette section les modes de défaillance d'un système interactif ainsi que leurs causes. Comme le rappelle la Figure 4.3, les défaillances (`failure`) sont provoquées par des fautes (`fault`) se propageant en erreur (`error`).

Figure 4.3. Chaîne de causalité des entraves à la sûreté de fonctionnement extraite des travaux de (Avizienis, Laprie, et al. 2004).

Pour déterminer les entraves à la sûreté de fonctionnement d'un système interactif, nous présentons tout d'abord le comportement générique d'un tel système. Nous présentons ensuite ses modes de défaillance, c'est-à-dire, ses dysfonctionnements possibles. Enfin, nous présentons le modèle de fautes étudié dans cette thèse, c'est-à-dire les fautes que nous considérons et qui peuvent provoquer les défaillances identifiées.

4.2.1 Comportement attendu

Le comportement d'un système interactif peut être résumé en deux buts du point de vue de l'utilisateur : traiter correctement ses actions en modifiant l'état du système de manière adaptée et lui présenter une représentation correcte de l'état du système et des actions qu'il peut effectuer. Ce comportement met en évidence deux flots d'événements et de données (voir Figure 4.4). Nous distinguons le traitement des actions effectuées par l'utilisateur (flot de contrôle) et le traitement des modifications de l'état interne du système (flot d'affichage) :

- *Le flot de contrôle* décrit l'ensemble du comportement du système, depuis la réception d'un événement utilisateur jusqu'à la modification de son état interne en fonction de l'action effectuée ;
- *Le flot d'affichage* décrit l'ensemble du comportement du système interactif, depuis la modification interne de son état jusqu'à la mise à jour de l'affichage proposé à l'utilisateur et représentant cet état.

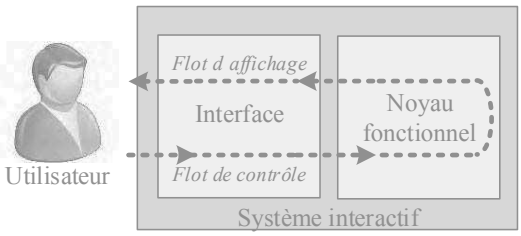


Figure 4.4. Flots de données dans les systèmes interactifs

4.2.2 Modes de défaillance

En considérant le comportement d'un système interactif décrit dans la section précédente, nous avons identifié quatre modes de défaillance :

- **Perte de contrôle :** le contrôle demandé par l'utilisateur n'est pas effectué.
- **Contrôle erroné :** le contrôle effectué ne correspond pas à celui demandé par l'utilisateur.
    - *Mauvais contrôle :* le contrôle effectué par le système est différent de celui qui aurait dû être réalisé compte tenu des actions de l'utilisateur.
    - *Contrôle spontané :* un contrôle est effectué par le système sans aucune action de l'utilisateur.

- **Perte d'affichage :** l'affichage correspondant au changement d'état du système n'est pas effectué. La perte d'affichage telle que nous la considérons correspond plus exactement à une perte du rafraichissement de l'affichage.
- **Affichage erroné :** l'affichage correspondant à l'état du système est effectué de manière inadéquate.
    - *Mauvais affichage :* l'affichage est effectué dans une mesure qui ne correspond pas à l'état du système.
    - *Affichage spontané :* l'affichage est modifié sans aucun changement d'état associé.

Comme le montre la Figure 4.5, l'identification des modes de défaillance fait partie entière du processus de certification des systèmes pour prouver la conformité à la norme CS 25.1309 (EASA 2014) qui définit les spécifications de sécurité-innocuité (safety) pour les systèmes embarqués dans les gros avions porteurs tels que l'Airbus A380.

(1) *General*. Compliance with the requirements of CS 25.1309(b) should be shown by analysis and, where necessary, by appropriate ground, flight, or simulator tests. Failure Conditions should be identified and their effects assessed. The maximum allowable probability of the occurrence of each Failure Condition is determined from the Failure Condition's effects, and when assessing the probabilities of Failure Conditions appropriate analysis considerations should be accounted for. Any analysis must consider:

Figure 4.5. Extrait des moyens de conformité avec la CS 25.1309 (extrait de la CS-25, AMC 25.1309-9.b (EASA 2014))

Les modes de défaillance proposés ci-dessus sont compatibles avec les recommandations de la partie AMC 25-11 de la CS-25 (voir Figure 4.6) qui donne des moyens de prouver la compatibilité des systèmes de contrôle-commande des avions gros porteurs avec la CS 25.1309. Ainsi, les pertes de contrôle et d'affichage rentrent dans la catégorie des `Loss of function` et les contrôles et affichages erronés rentrent dans la catégorie des `Malfunction`.

(1) The type of display system failure conditions will depend, to a large extent, on the architecture (Integrated Modular Avionics, Federated System, Non-Federated System, etc.), design philosophy, and implementation of the system. Types of failure conditions include:

- Loss of function (system or display).
- Failure of display controls – loss of function or malfunction such that controls perform in an inappropriate manner, including erroneous display control.
- Malfunction (system or display) that leads to:
    - Partial loss of data, or
    - Erroneous display of data that is either:
        - Detected by the system (for example, flagged or comparator alert), and/or easily detectable by the flight crew; or
        - Difficult to detect by the flight crew or not detectable and assumed to be correct (for example, "Misleading display of ….").

Figure 4.6. Identification des modes de défaillance pour les systèmes de commande-contrôle (extrait de la CS-25, AMC 25-11-Chapitre 4-21.a (EASA 2014))

Les modes de défaillance présentés dans cette thèse permettent la classification des modes de défaillance pour les systèmes interactifs. Il est important de noter que ceux-ci doivent être détaillés en fonction des systèmes concernés. Par exemple, la conséquence d'une perte de contrôle n'aura pas le même impact s'il s'agit d'un contrôle concernant le système de conditionnement d'air ou d'un contrôle concernant l'autopilote.

Dans les sous-sections suivantes, nous détaillons les quatre modes de défaillance identifiés en tenant compte des hypothèses et du périmètre de la thèse. Nous considérons ici les hypothèses H1, H3, H4 et H5 qui concentrent le périmètre de la thèse sur les composants logiciels de la partie interface du système interactif (les autres composants du système étant fiables d'après les hypothèses). C'est donc cette partie que nous voulons fiabiliser et pour cela, nous détaillons ci-dessous les modes de défaillance identifiés à son niveau.

Perte de contrôle : le contrôle demandé par l'utilisateur n'est pas effectué.

Dans ce cas, l'interface n'a pas traité correctement les événements correspondants aux actions de l'utilisateur sur les périphériques d'entrée et n'a pas envoyé les événements correspondant à ces actions au noyau fonctionnel (voir Figure 4.7).

Exemple : *L'utilisateur effectue un click sur le bouton permettant d'engager un changement de cap de manière à déclencher le contrôle correspondant. Les événements correspondants à cette action n'ont pas été envoyés au noyau fonctionnel impliquant que le contrôle désiré par l'utilisateur n'est pas pris en compte par le système interactif : l'engagement de changement de cap n'est pas pris en compte et donc pas effectué.*

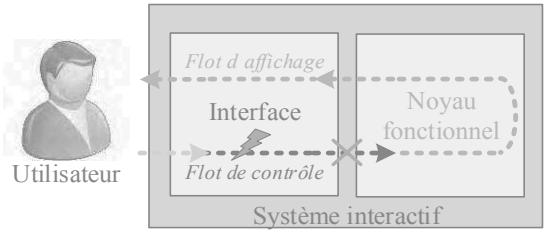


Figure 4.7.Comportement d'un système interactif lors d'une perte de contrôle

Contrôle erroné : le contrôle effectué ne correspond pas à celui demandé par l'utilisateur. Il est important de noter que dans ce cas, nous considérons que le contrôle erroné est envoyé au bon noyau focntionnel.

Mauvais contrôle : (voir Figure 4.8-a)

Le contrôle effectué par le système est différent de celui désiré par l'utilisateur. Dans ce cas, l'interface n'a pas traité correctement les événements correspondants aux actions de l'utilisateur sur les périphériques d'entrée et a envoyé au noyau fonctionnel des événements correspondant soit à une autre action, soit à la même action avec une valeur différente que celle choisie par l'utilisateur.

Exemple 1 : *L'utilisateur clique sur le bouton permettant d'engager un changement de cap pour enclencher le contrôle correspondant. Cette action est prise en compte par l'interface de manière erronée : celle-ci envoie un événement correspondant à une autre action utilisateur au noyau fonctionnel. Le noyau fonctionnel effectue un contrôle erroné : l'engagement de changement de cap n'est pas effectué et le système effectue un autre contrôle, par exemple le désengagement de l'autopilote.*

Exemple 2 : *Après avoir rentré une valeur de 30° dans l'EditBoxNumeric correspondant à la valeur du cap, l'utilisateur la valide en tapant la touche de validation. Cette action est prise en compte par l'interface qui envoie l'événement correspondant à la validation du changement de valeur de cap mais avec une valeur erronée (90°). Le système interactif traite donc ce contrôle avec une valeur erronée : la valeur du cap est modifiée avec une valeur de 90° au lieu de la valeur de 30° désirée par l'utilisateur.*

Contrôle spontané : (voir Figure 4.8-b)

Un contrôle est effectué par le système sans aucune action de l'utilisateur. Dans ce cas, l'interface va envoyer au noyau fonctionnel des événements correspondant à une ou plusieurs actions de l'utilisateur sans que celui-ci n'en ait effectué aucune.

Exemple : *L'interface envoie des événements au noyau fonctionnel sans action de l'utilisateur. Ce qui a pour conséquence que le système désengage l'autopilote sans que l'utilisateur ne l'ait demandé et sans qu'il n'ait effectué d'action sur les périphériques d'entrée.*

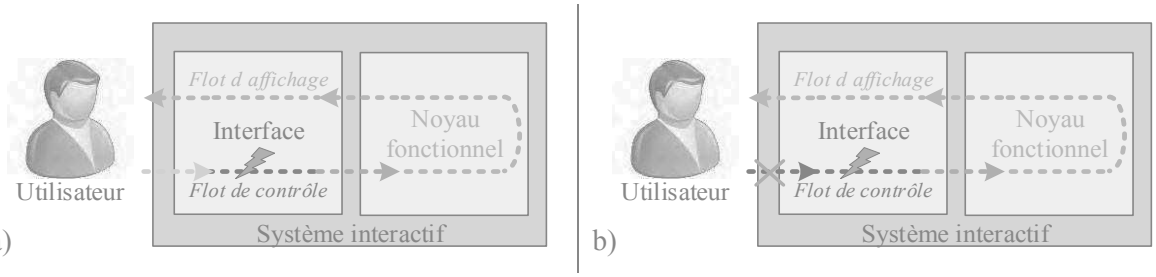


Figure 4.8. Comportement d'un système interactif lors d'un contrôle erroné

Perte d'affichage : l'affichage correspondant au changement d'état du système n'est pas effectué.

Dans ce cas, l'interface n'a pas tenu compte des informations provenant du noyau fonctionnel et attestant de son changement d'état et n'a pas effectué la modification de l'affichage en conséquence (voir Figure 4.9).

Exemple : *L'autopilote notifie son désengagement, ce qui nécessite une modification de l'affichage. Cette modification n'est pas effectuée par l'interface de manière à ce que l'autopilote soit toujours montré actif sur l'écran. Ainsi, le pilote ne peut pas prendre connaissance de ce changement d'état.*

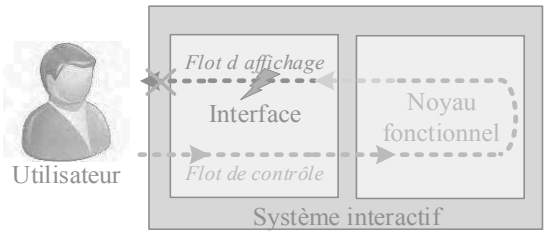


Figure 4.9. Comportement d'un système interactif lors d'une perte d'affichage

Affichage erroné : l'affichage correspondant à l'état du système est effectué de manière inadéquate.

Mauvais affichage : (voir Figure 4.10-a)

L'affichage est effectué d'une manière qui ne correspond pas à l'état du système. Dans ce cas, l'interface modifie son affichage mais de manière à ce que celui-ci ne corresponde pas à l'état réel du noyau fonctionnel.

Exemple : *L'autopilote notifie son désengagement, ce qui nécessite une modification de l'affichage. Suite à cette notification, l'interface effectue un changement d'affichage ne lui correspondant pas (l'écran affiche un changement de cap).*

Affichage autonome : (voir Figure 4.10-b)

L'affichage est modifié sans aucun changement d'état associé. Dans ce cas, l'interface modifie son affichage sans avoir reçu de notification du noyau fonctionnel.

Exemple : *Sans notification du noyau fonctionnel, l'interface affiche un désengagement de l'autopilote.*

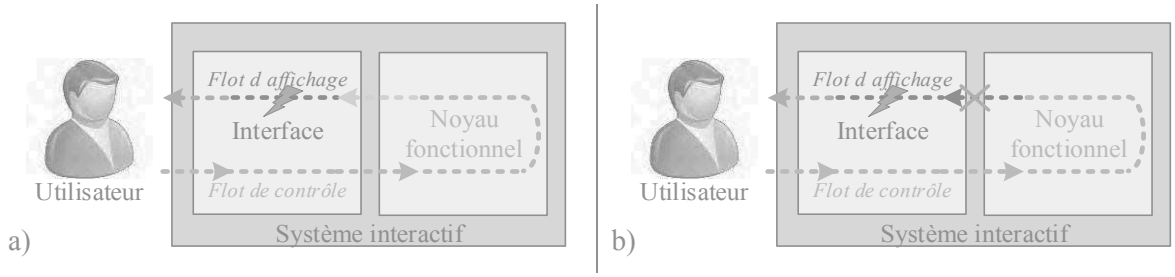


Figure 4.10. Comportement d'un système interactif lors d'un affichage erroné

4.2.3 Modèle de fautes

Nous venons de définir les défaillances des systèmes interactifs que nous chercherons à éviter grâce à l'approche proposée dans cette thèse. Nous déterminons maintenant les causes possibles de telles défaillances afin de pouvoir mettre en place des méthodes pour les prévenir et les traiter. Comme rappelé en Figure 4.3, les causes primaires de défaillance sont les fautes qui peuvent affecter le système. Pour identifier, le modèle de faute de cette thèse, c'est-à-dire les fautes que nous considérons, nous reprenons la classification des fautes proposée par (Avizienis, Laprie, et al. 2004) (voir section 2.3). Les hypothèses H1, H2 et H6 nous permettent d'éliminer respectivement les erreurs humaines en opération, les fautes malveillantes et les fautes matérielles de développement.

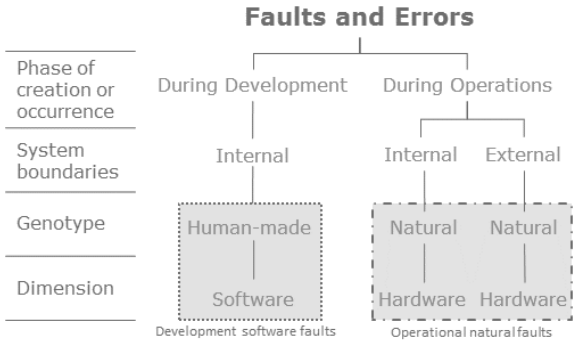


Figure 4.11. Classes de fautes considérées dans la thèse

Notre modèle de faute comprend donc deux classes de fautes (représentées en Figure 4.11) :

Les fautes logicielles de développement :

Ce sont les fautes introduites pendant le développement du système interactif. Elles sont d'origine humaine et peuvent être introduites pendant toutes les phases de développement du logiciel, depuis la spécification du système jusqu'à son codage. Elles peuvent ainsi aller de mauvaises spécifications à des erreurs de codage telles que des confusions de noms de variables, des problèmes d'espace mémoire… Ces fautes sont considérées dans la littérature comme des défauts du logiciel et ils ont notamment été étudiés et classifiés par (Sullivan et Chillarege 1991) et (Chillarege, et al. 1992). Plus particulièrement, les travaux de (Lelli, Blouin et Benoit 2015) classifient les fautes logicielles de développement qui peuvent affecter les systèmes interactifs. De plus, des études telles que (Basili et Perricone 1984) ont montré que leurs occurrences augmentent avec la complexité et la taille des systèmes.

Bien que ces fautes soient connues et étudiées depuis longtemps, elles sont toujours un défi pour la sûreté de fonctionnement des systèmes informatiques et ce notamment du fait de l'accroissement constant de la complexité du logiciel. Ainsi, si l'on prend l'exemple du contexte applicatif de cette thèse (l'avionique), on peut remarquer une croissance fulgurante du nombre de lignes de codes embarquées dans les avions (douze millions pour l'A320 contre plus de soixante-cinq millions pour l'A380) (Arlat, et al. 2006).

Les fautes naturelles en opération :

Ce sont les fautes causées par un phénomène naturel, sans aucune participation humaine, affectant le matériel et donc par propagation, pouvant affecter le logiciel du système. Ces fautes peuvent être des fautes de crash (d'arrêt du système) dues par exemple à la défaillance de la source d'énergie du système. Elles peuvent également être des fautes beaucoup plus subtiles telles que des inversions d'un ou plusieurs bits (appelés single ou multiple event upsets - SEU & MEU) causées par exemple par des rayonnements électromagnétiques ou d'autres modifications de mémoire (appelées single event effects - SEE).

La considération de ces fautes est d'autant plus justifiée par le contexte aéronautique de la thèse. En effet, de nombreuses études telles que (Taber et Normand 1993) pour les SEU et (Normand 1996b) pour les SEE, montrent que ces phénomènes ne sont pas négligeables dans les systèmes avioniques, ces fautes étant fréquentes aux altitudes où volent les avions (Ziegler et Lanford 1979). Cependant, des études ont également montré que ces fautes pouvaient également apparaître au niveau du sol terrestre (Normand 1996a).

Il est important de noter que certaines de ces fautes, en particulier les SEU, sont généralement traitées par des codes correcteurs d'erreurs (appelés ECC pour Error Correcting Code) tels que les CRC (pour Cyclic Redundancy Code (Koopman et Chakravarty 2004)). Ces codes correcteurs d'erreur sont souvent appliqués au niveau du matériel et sont embarqués au niveau des mémoires et des caches des systèmes informatiques critiques. Cependant, comme nous le montrent les travaux de (Koopman et Chakravarty 2004), même si ces ECC peuvent être capables de détecter des MEU, ils possèdent tous

une distance de Hamming, nombre représentant le nombre de bits altérés que peut corriger l'ECC et ne sont donc par conséquent pas capable de détecter toutes les fautes naturelles.

De plus, l'évolution des composants matériels dans les avions justifie également fortement la considération de ces fautes car elle conduit à une augmentation des probabilités de leurs apparitions du fait de la miniaturisation du matériel. Ainsi, les recommandations dans l'avionique sont de 100 fautes sur la durée de vie de l'avion (plus de 25 ans), ce qui est le cas pour les composants matériels utilisés à l'heure actuelle dans les avions. Cependant, les composants matériels modernes (que l'on pourrait utiliser à terme dans l'avionique) affichent un taux de 1000 fautes sur une durée de vie de 1 à 5 ans (ce taux augmenterait donc considérablement si l'on prenait en compte la durée de vie d'un avion) (Regis, et al. 2013).

4.3 Une architecture générique pour les systèmes interactifs

4.3.1 Principe et rationnel

Notre objectif est de proposer une architecture logicielle générique pour les systèmes interactifs de type WIMP identifiant leurs différents composants fonctionnels. La définition d'une telle architecture fait partie des processus de développement des systèmes interactifs critiques comme celui proposé par (Ladry 2010). Une telle architecture permet de faciliter le développement des systèmes interactifs et participe à leur fiabilisation (objectif fixé dans cette thèse). La décomposition fonctionnelle de notre architecture permet d'identifier des composants sur lesquels nous pourrons appliquer notre approche (présentée en section 4.4). En effet, ces composants pourront être modélisés et analysés de façon formelle (premier axe de l'approche). Nous pourrons également leur appliquer des mécanismes de tolérance aux fautes (deuxième axe de l'approche).

Cette architecture est générique en deux points. Premièrement, elle est générique au niveau de la technique d'interaction (toute application interactive WIMP peut être développée selon cette architecture dans la mesure où les périphériques d'entrée sortie sont un clavier, un désignateur graphique et un écran). Deuxièmement, elle est générique par rapport à toutes les applications interactives du cockpit qui la respectent. Par conséquent, elle est également générique avec tous les cockpits d'avions compatibles avec la norme ARINC 661.

Pour concevoir cette architecture, nous avons choisi de nous appuyer sur le modèle architectural ARCH (Bass, et al. 1992) (voir section 2.4.1). En effet, ce modèle propose une décomposition fonctionnelle en cinq briques (de gauche à droite sur la Figure 4.2 : le noyau fonctionnel et son adaptateur, le contrôleur de dialogue, l'interaction logique ou présentation et l'interaction physique ou boite à outils qui permet d'être générique.

Il est important de noter que l'architecture proposée dans cette section est très détaillée pour tous les niveaux de l'interaction (des périphériques d'entrée et de sortie jusqu'au contrôleur de dialogue). Il est vrai que ce niveau de détail n'est pas nécessaire pour un certain nombre d'applications interactives. En effet, si nous prenons l'exemple des applications interactives développées pour être intégrés dans un système d'exploitation comprenant un window manager chargé de la gestion des entrées et des sorties, certains composants de l'architecture n'ont pas à être développés. Cependant, le contexte de cette thèse nous impose de détailler cette architecture sur tous les niveaux de l'interaction. En effet, dans le cas des cockpits interactifs, tous les composants de l'architecture présentée (des périphériques d'entrée et de sortie jusqu'au contrôleur de dialogue) devront être développés de manière à pouvoir être certifiés et être ainsi intégrés dans le cockpit.

4.3.2 Description

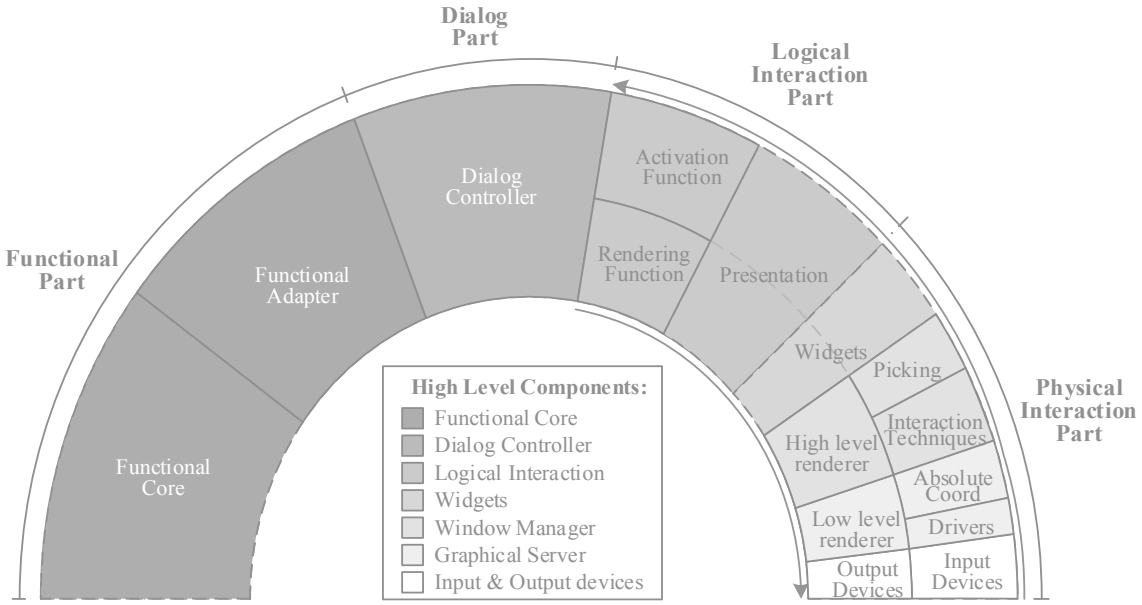


Figure 4.12. Les composants de notre architecture et leur correspondance avec le modèle architectural ARCH

La Figure 4.12 présente les différents composants de l'architecture que nous proposons et leur correspondance avec le modèle architectural ARCH (Bass, et al. 1992). Nous nous intéressons dans cette thèse à l'interface du système interactif et non à son noyau fonctionnel. C'est pour cette raison que nous ne détaillons pas dans l'architecture proposée en Figure 4.12 la partie dédiée au noyau fonctionnel (`Functional Part`), composée du noyau fonctionnel (`Functional Core`) et son adaptateur (`Functional Adapter`) qui permet de traduire les données provenant du noyau fonctionnel en données compréhensibles par le contrôleur de dialogue, et inversement.

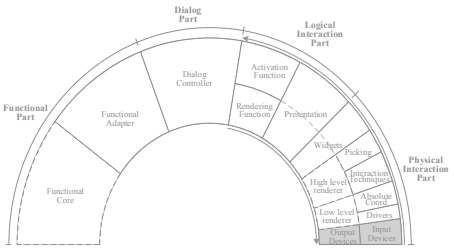
Nous proposons de diviser la partie interface du système (composée du contrôleur de dialogue, des interactions logiques et des interactions physiques) en douze composants, regroupés en sept composants principaux que nous détaillons dans les sous-sections suivantes. Nous ne présentons dans cette section qu'une description haut-niveau des différents composants identifiés afin de rester générique. En effet, pour décrire les composants à un plus bas niveau, il faudrait instancier l'architecture en donnant par exemple les périphériques d'entrée-sortie utilisés, l'application, …

4.3.2.1 L'interaction physique

Cette partie est responsable de l'interaction physique avec l'utilisateur final au travers des composants logiciels et matériels. Le niveau matériel est constitué des périphériques d'entrée-sortie (`Output Devices` & `Input Devices`). Au niveau logiciel, cette partie a la responsabilité de la récupération des événements de l'utilisateur et leur transmission aux objets d'interaction (les objets graphiques avec lesquels l'utilisateur peut interagir et qui servent de liaison entre l'interaction physique et l'interaction logicielle). Elle a également la responsabilité de la mise à jour du rendu de l'application. Elle peut être divisée en quatre composants principaux que nous détaillons dans les sous-sections suivantes.

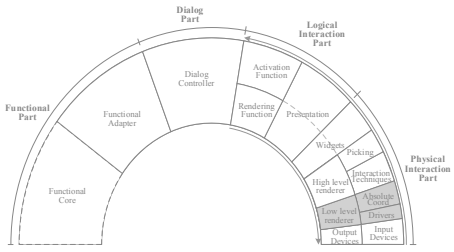
Les périphériques d'entrée et de sortie

Ce sont les composants matériels, les périphériques physiques permettant l'interaction. Dans le cas des systèmes interactifs de type WIMP, les *périphériques de sortie* (`Output Devices`) sont généralement un écran et éventuellement des haut-parleurs. Les *périphériques d'entrée* (`Input Devices`) quand à eux sont généralement un clavier et un périphérique



de pointage (souris, touchpad, trackball…). Ces périphériques enregistrent les actions de l'utilisateur.

Le serveur graphique



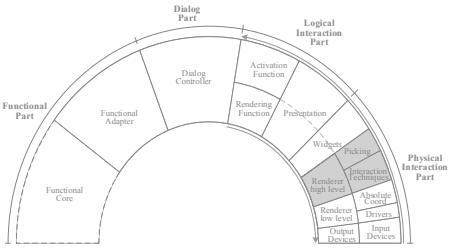
Le *serveur graphique* est composé des drivers, du composant chargé du calcul des coordonnées absolues du widget et du composant de rendu bas-niveau.

Les *drivers* sont les composants logiciels en charge de récupérer les événements bas-niveau en provenance des périphériques d'entrées. Ce composant est donc chargé de contrôler périodiquement l'état du périphérique d'entrée et de produire les événements correspondant à son état. Si l'on prend l'exemple d'une souris, cela correspond à l'état de ses boutons (pressés ou relâchés) et à son déplacement relatif par rapport à sa position précédente.

Le composant en charge du calcul des coordonnées absolues (`Absolute Coord`) reçoit les événements des drivers, autrement dit, les événements comprenant la position relative du curseur et met à jour une variable contenant la position absolue du curseur. Il envoie ensuite les événements au composant suivant (`Interaction Techniques`).

Le composant rendu bas-niveau (`Low lever renderer`) a la responsabilité de la transformation des événements de rendu en commandes graphiques exécutables interprétables par le périphérique de sortie qui les exécutera. Il a également la responsabilité du calcul du feedback immédiat, c'est-à-dire, de la réponse immédiate du système interactif aux actions de l'utilisateur, par exemple la représentation graphique du déplacement du curseur sur l'écran.

Le window manager

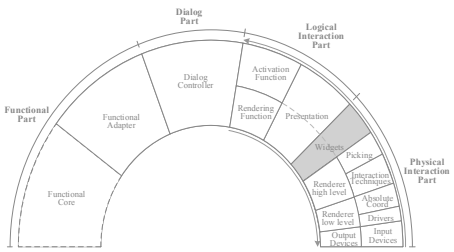


Le *window manager* est composé du composant permettant le calcul des différentes techniques d'interaction (`Interaction Techniques`), du composant responsable du picking (`Picking` : identification du widget concerné par l'action de l'utilisateur) et du composant permettant le rendu de haut-niveau.

Le composant `Interaction Techniques`, en charge du calcul des techniques d'interaction a la responsabilité de la création d'événements haut-niveau en combinant les événements bas-niveau venant du composant `Absolute Coord`. En reprenant l'exemple d'une souris, nous pouvons facilement décrire l'événement `click` qui est la combinaison de deux événements souris consécutifs (pressé puis relâché) sans que la souris n'ait considérablement bougé et dans un laps de temps limité.

Le composant `Picking` est en charge de la connexion entre les événements de haut-niveau provenant du composant `Interaction Techniques` et les objets graphiques responsables de l'interaction. Ainsi, ce composant est en charge de trouver l'objet graphique présent sous le curseur de la souris (par conséquent, l'objet graphique avec lequel l'utilisateur souhaite interagir) et de lui transférer les événements de haut-niveau qui lui sont destinés (par exemple, un clic qui correspond à la combinaison d'une pression et d'un relâchement du bouton de la souris).

Les widgets



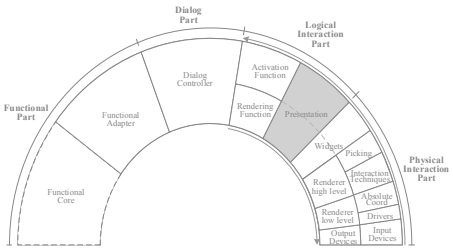
Les *widgets* sont les composants graphiques avec lesquels l'utilisateur peut interagir. L'ensemble des widgets est divisé en plusieurs ensembles différents correspondant chacun à une fenêtre et une application. Ils informent la fonction de picking de leur modification d'état (par exemple lorsqu'un bouton est désactivé, il ne pourra pas être sélectionné par l'utilisateur). Ils informent également la fonction de rendu de haut-niveau de leur changement d'état. En effet, celle-ci est abonnée à leur

changement d'état de manière à ce que celle-ci puisse faire les modifications d'affichage correspondantes (par exemple, le changement de couleur d'un texte).

4.3.2.2 L'interaction logique

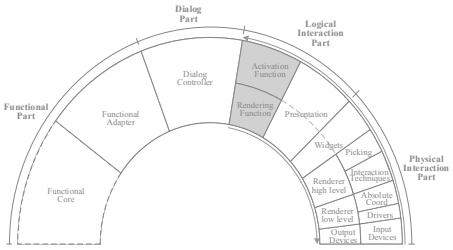
Cette partie a la responsabilité de la conversion des informations provenant des widgets en des informations abstraites qui seront transmises au contrôleur de dialogue (`Dialog Controller`). Elle est également responsable de la récupération des informations du contrôleur de dialogue et de leur transformation en informations compréhensibles par les widgets. Elle est divisée en trois composants : la présentation et les fonctions d'activation et de rendu.

La présentation



La *présentation* est un composant qui récupère les événements provenant des widgets pour les envoyer à la fonction d'activation et qui transforme également les événements provenant des fonctions d'activation et de rendu de manière à les transformer en événements entraînant la modification de l'état interne des widgets. C'est l'unique composant de l'interaction logique qui est en communication avec les widgets. Il a ainsi l'avantage de proposer une interface unique aux widgets et ainsi d'augmenter la modifiabilité de l'application (notamment en termes de changement de widgets). Par exemple, la présentation transformera un événement `click` sur un bouton en un événement de plus haut-niveau qui sera interprété par la fonction d'activation. Ainsi, si l'on désire remplacer le bouton en question par un autre widget, il suffira de modifier ce composant sans avoir à modifier tous les composants de l'interaction logique.

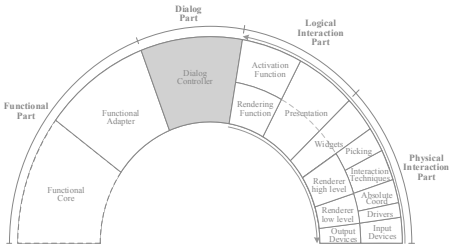
Les fonctions d'activation et de rendu



La *fonction d'activation* est responsable du déclenchement des événements sur le contrôleur de dialogue en fonction des événements provenant de la présentation. Elle permet également, au travers de la présentation, de faire le lien entre les disponibilités des widgets et l'état du contrôleur de dialogue.

La *fonction de rendu* met en relation l'état du contrôleur de dialogue avec l'information présentée à l'utilisateur par les widgets au travers de la présentation (par exemple, suivant l'état du contrôleur de dialogue, elle peut demander l'affichage d'un groupe de widgets plutôt qu'un autre).

4.3.2.3 Le contrôleur de dialogue



Cette partie est responsable du séquencement des tâches du système interactif. Elle décrit précisément, en fonction de l'état du noyau fonctionnel, l'ensemble des actions pouvant être effectuées par l'utilisateur. Elle est aussi responsable de la répercussion de l'effet de l'exécution d'une action par l'utilisateur sur l'état du noyau fonctionnel et sur l'état de l'interface.

4.3.3 Description AADL

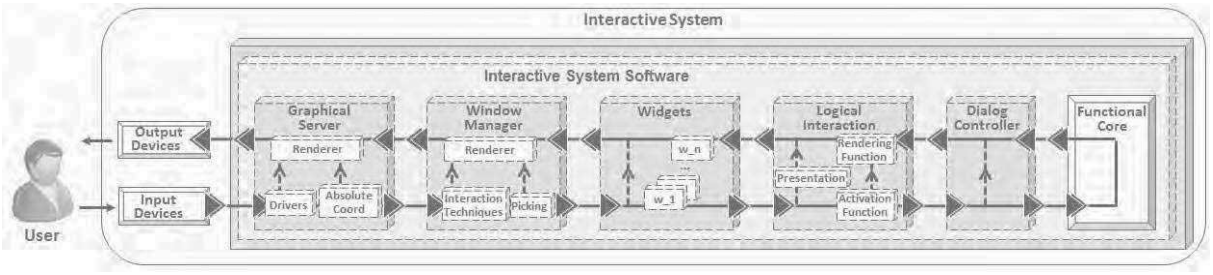


Figure 4.13. Description AADL de notre architecture

Afin d'expliciter les liaisons entre les différents composants présentés, nous présentons en Figure 4.13 la description AADL de notre architecture à l'aide du langage architectural graphique AADL (SAE International 2012) (voir section 2.4.3). Par souci de lisibilité, nous avons décidé de placer l'utilisateur ainsi que les périphériques d'entrée-sortie à gauche et le noyau fonctionnel à droite, inversant ainsi le sens proposé par le modèle architectural ARCH. Cette configuration permet une lecture de gauche à droite lorsque l'on s'intéresse à l'utilisation du système par un opérateur. Sur cette architecture, nous retrouvons nos composants présentés en Figure 4.12 auxquels nous avons adjoint l'utilisateur :

- Les périphériques d'entrée et de sortie, représentés comme des périphériques matériels ;
- Le serveur graphique, constitué des drivers, du composant en charge du calcul des coordonnées absolues et du composant de rendu bas-niveau ;
- Le gestionnaire de fenêtre, constitué du composant en charge du calcul des techniques d'interaction, du composant en charge du picking et du composant de rendu haut-niveau ;
- L'ensemble des widgets du système ;
- L'interaction logique, constitué de la présentation, la fonction d'activation et la fonction de rendu ;
- Le contrôleur de dialogue ;
- Le noyau fonctionnel.

Cette représentation de l'architecture permet de mettre en évidence les liens de communication entre les composants haut-niveau de notre architecture. Pour des raisons de lisibilité, nous ne détaillons pas ici les liens entre les composants bas-niveau. De plus, ces liens ne peuvent pas être décrits précisément car ils dépendent beaucoup de l'instanciation de l'architecture.

4.3.4 Instanciation pour les cockpits interactifs

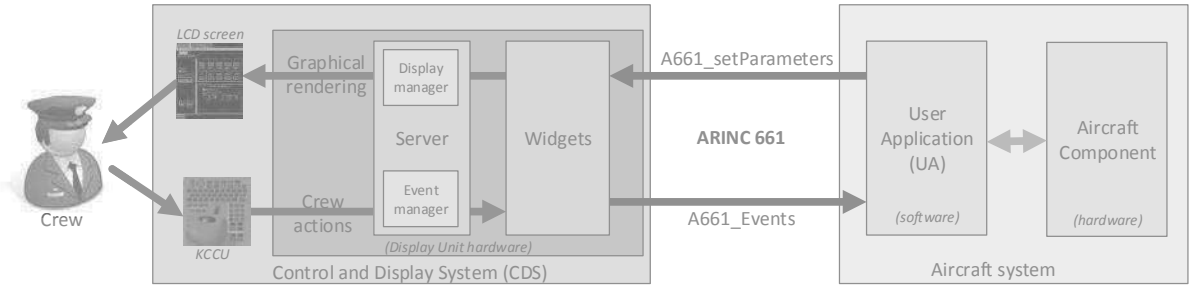


Figure 4.14. Architecture logicielle et matérielle d'un cockpit interactif respectant le standard ARINC 661

Les systèmes interactifs dans les avions doivent répondre à un certain nombre de normes et standards pour pouvoir être certifiés et être embarqués dans l'avion. Le standard ARINC 661 définit notamment l'architecture qu'ils doivent respecter, rappelée en Figure 4.14 (voir section 3.1.2).

La difficulté ici est d'instancier l'architecture générique présentée dans la section 4.3.2 tout en respectant l'architecture définie par le standard ARINC 661. Une telle instanciation est présentée en

Figure 4.15. Pour simplifier la figure, le CDS représenté ici n'interagit qu'avec un seul pilote (`Pilot`) et un seul système avion (`AircraftSystem`). La composition du CDS est également simplifiée car nous n'avons représenté ici qu'une seule DU (`DisplayUnitSystem`) et un seul KCCU (Keyboard and Cursor Control Unit).

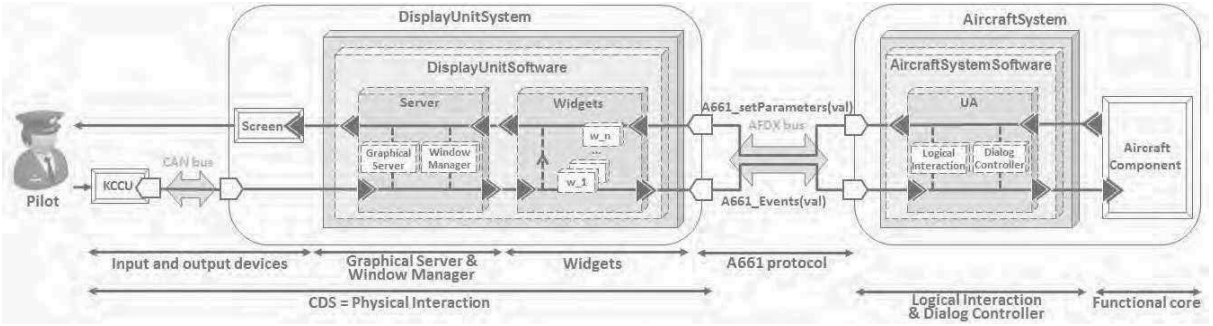


Figure 4.15. Architecture simplifiée d'un cockpit interactif respectant le standard ARINC 661

La Figure 4.15 présente les liens entre les composants définis par le standard ARINC 661 et les composants définis dans notre architecture générique. Par souci de lisibilité, nous n'avons représenté ici que les composants haut-niveau. Nous distinguons donc :

- *Les périphériques d'entrée et de sortie :* le KCCU et l'écran, qui permettent l'interaction entre le système et l'équipage.
- *Le serveur :* il est composé du serveur graphique et du gestionnaire de fenêtre. Il est donc responsable de la gestion des évènements provenant du KCCU (`Drivers`, `Absolute Coord`, `Interaction Techniques` et `Picking`) et du rendu graphique de l'application sur l'écran (`Renderer`).
    - Le composant de calcul des coordonnées absolues (`Absolute Coord`) présente en plus un comportement spécifique pour le curseur : il offre aux widgets un mode d'emprisonnement (`Caging`) dans lequel le curseur disparait de manière à ce que l'interaction ne puisse plus se faire qu'avec un seul widget (pour pouvoir concentrer l'attention de l'utilisateur sur cette interaction).
    - Le composant en charge du calcul des techniques d'interaction (`Interaction Techniques`) comprend une seule technique d'interaction qui est le clic : il est responsable de la définition des clics provenant du KCCU à partir des événements que celui-ci fournit (bouton pressé et bouton relâché).
    - Le `Picking` est en charge de l'identification des widgets ciblés par les actions utilisateurs.
    - Le serveur contient également un composant appelé graphe de scène (`Scene Graph`) qui est responsable de la gestion de la hiérarchie de widgets. C'est ce composant qui regroupe toutes les caractéristiques de l'ensemble des widgets nécessaires à la fois pour le rendu et le picking.
    - Le composant de rendu (`Renderer`) gère l'affichage général de l'application ainsi que celui du curseur graphique correspondant à la trackball du KCCU. Il est donc abonné aux modifications d'états des composants `Absolute Coord` et `Scene Graph`.
- *Les widgets :* chaque User Application (UA), autrement dit la partie logicielle d'un système avionique est liée à un ensemble de widgets organisés de manière hiérarchique dans un graphe de scène géré par le serveur. Ces widgets correspondent aux éléments d'affichage et d'interaction d'une UA.
- *Les systèmes avioniques :* ils sont composés des différents systèmes physiques de l'avion (`AircraftComponent`) (*e.g.* les moteurs) ainsi que d'une interface logicielle l'User Application (`UA`). L'UA permet de traiter les actions de l'équipage sur les widgets via la réception d'événements conformes au standard ARINC 661 (`A661_Events(val)`) envoyés par le CDS. Le CDS permet également à l'UA de notifier l'équipage d'un changement d'état du système avionique à l'aide de méthodes `A661_setParameters(val)` permettant de modifier l'affichage et l'état des widgets. L'UA correspond à la combinaison de l'interaction logique et du contrôleur de dialogue. Elle est donc

constituée des quatre composants présentés dans la définition de notre architecture (la présentation, les fonctions d'activation et de rendu et le contrôleur de dialogue).

Cette architecture nous permet d'identifier deux parties dans les composants logiciels de ce type de systèmes interactifs : une partie générique composée du serveur et des widgets et une partie spécifique composée de l'UA, dépendant très fortement du système avionique contrôlé par le système interactif.

4.3.5 Avantages

La décomposition d'un système interactif en composants respectant le modèle architectural ARCH présente six avantages que nous explicitons ci-dessous.

- *Réutilisabilité* : le fait de découper le système interactif en composants de bas niveau permet de réutiliser les composants génériques, c'est-à-dire non spécifiques à une application en particulier, dans le développement d'une nouvelle application. Ainsi, les composants de l'interaction physique ne sont pas dépendants de l'application et peuvent être réutilisés tels quels dans une autre application possédant les mêmes périphériques d'entrée et de sortie.
- *Modifiabilité* : du fait de la définition de composants bas-niveau, il est également simple de modifier une partie du système interactif sans avoir à tout modifier. Il est ainsi possible d'imaginer remplacer un périphérique d'entrée par un autre : il sera nécessaire de revoir le driver correspondant. Toutefois, si celui-ci fournit les mêmes événements, il n'y aura pas besoin de revoir les composants de plus haut-niveau.
- *Lisibilité* : la décomposition de l'architecture en petits composants permet de limiter leur complexité, il est donc plus facile de comprendre leur comportement et de le présenter à des personnes tierces.
- *Analyse* : la décomposition de l'architecture en petits composants la rend plus facilement analysable. Ainsi, un composant plus petit pourra mieux être vérifié et sera donc un meilleur candidat à la conception zéro-défaut que nous présentons dans le chapitre suivant.
- *Généricité* : bien que notre architecture ne présente pas spécifiquement la prise en compte des spécificités des systèmes interactifs post-WIMP (qui sont hors du périmètre de la thèse), sa généricité permet de les prendre en compte. D'autres travaux les ont d'ailleurs pris en compte sur des architectures similaires. Ainsi, dans (Hamon, Palanque, et al. 2013) les auteurs proposent un raffinement du modèle architectural ARCH pour permettre la prise en compte des interactions multi-touch ; dans (Bastide, Navarre, et al. 2004), les auteurs s'intéressent à une approche à base de modèles, fondée sur le modèle architectural ARCH pour développer les systèmes multimodaux. Les aspects multimodaux sont d'autant plus intéressants lorsque l'on considère les défaillances possibles des périphériques d'entrée et de sortie ; ainsi, dans (D. Navarre, P. Palanque, et al. 2008) les auteurs proposent une architecture et une technique de description formelle pour le développement de systèmes reconfigurables suite à la défaillance des périphériques d'entrée ou de sortie.
- *Fournit un support à la certification* : la définition de l'architecture logicielle d'un système fait partie des processus de développement recommandés pour la certification tel que celui de la norme DO-178C (RTCA et EUROCAE 2011a). Cette norme recommande en effet de définir le design du logiciel à l'aide d'exigences bas-niveau et d'une architecture logicielle.

4.4 Une approche pour des systèmes interactifs sûrs de fonctionnement

L'objectif de cette thèse est de proposer une méthode et des outils pour spécifier et développer des systèmes interactifs sûrs de fonctionnement. Nous avons identifié en section 4.2.2 les défaillances des systèmes interactifs que nous cherchons à éviter et leurs causes ont été définies par notre modèle de fautes en section 4.2.3. Notre objectif est donc de rendre les systèmes interactifs robustes par rapport aux fautes logicielles de développement et aux fautes naturelles en opération. Pour cela, nous proposons une approche globale composée d'une approche à base de modèles pour concevoir des systèmes interactifs zéro-défaut en prévenant et éliminant les fautes logicielles de développement et d'une

architecture logicielle générique et tolérante aux fautes permettant de tolérer les fautes matérielles en opération. Nous proposons également des méthodes et outils pour la mise en œuvre de notre approche.

4.4.1 Approche à base de modèles pour des systèmes interactifs zéro-défaut

Dans un premier temps, nous proposons de concevoir et développer les systèmes interactifs en prévenant et éliminant les fautes logicielles de développement. Pour cela, nous proposons un développement à base de modèles de manière à développer un logiciel zéro-défaut. Un logiciel zéro-défaut (Hamilton 1986) est un logiciel ne contenant aucune faute de développement logiciel et se comportant exactement comme attendu. Nous proposons l'utilisation d'un langage de description formel pour la modélisation du comportement de chaque composant de notre architecture. Le langage que nous utilisons est appelé ICO (pour Interactive Cooperative Objects, (D. Navarre, P. Palanque, et al. 2009)), il est dédié à la description formelle des systèmes interactifs. ICO permet de décrire le comportement de composants interactifs de manière concise, complète et non-ambigüe. Ce langage est fondé sur les principes des réseaux de Petri haut-niveau (Genrich 1991) ce qui permet d'analyser formellement les modèles. Ainsi, la modélisation des composants de notre architecture permet de prévenir l'introduction de fautes logicielles de développement dans les systèmes interactifs et l'analyse formelle de ces modèles permet l'élimination des fautes résiduelles. Cette méthode est détaillée dans le Chapitre 5.

4.4.2 Architecture logicielle générique pour des systèmes interactifs tolérants aux fautes

Dans un second temps, nous proposons d'intégrer des mécanismes de tolérance aux fautes dans les systèmes interactifs. En effet, notre modèle de fautes comprend les fautes matérielles en opération. Ces fautes sont inévitables et imprévisibles et ne peuvent donc pas être traitées par prévention et élimination comme les fautes logicielles de développement. Des études ont prouvé que ce genre de fautes n'était pas négligeable, surtout dans l'avionique (Normand 1996b) et (Taber et Normand 1993). Le seul moyen de traiter ces fautes est donc de les tolérer (voir section 2.3.4). Nous proposons donc l'ajout d'un mécanisme de tolérance aux fautes aux composants de notre architecture. Nous avons choisi pour cela le mécanisme de programmation n-autotestable (Laprie, Arlat, et al. 1990). Ce mécanisme nous permet de détecter les erreurs provoquées par les fautes naturelles et de remettre le système dans un état fonctionnel après leur apparition et il est utilisé pour les commandes de vol électriques des avions Airbus depuis l'A320. Les raisons du choix d'un tel mécanisme et notre méthode pour l'appliquer aux composants logiciels des systèmes interactifs sont développées dans le Chapitre 6.

4.4.3 Des outils pour la mise en œuvre de l'approche

Nous présentons au Chapitre 7 des moyens pour permettre la mise en œuvre de notre approche. Ainsi, nous proposons l'utilisation d'un outil permettant mise en œuvre de la conception zéro-défaut. Cet outil, nommé PetShop (Navarre 2001) permet la modélisation, l'exécution et l'analyse des modèles ICO. Nous proposons également ARISSIM, un simulateur de noyau avionique ARINC 653 ainsi qu'une maquette pour mettre en œuvre l'introduction de mécanismes de tolérance aux fautes dans les systèmes interactifs.

4.5 Conclusion

Nous avons présenté dans ce chapitre le périmètre et les hypothèses de cette thèse. Ainsi, nous nous intéressons dans cette thèse à la sûreté de fonctionnement des systèmes interactifs et plus particulièrement à celle de leurs composants logiciels responsables des fonctionnalités interactives.

Nous avons ensuite présenté les modes de défaillances des systèmes interactifs que nous cherchons à éviter et le modèle de faute associé, constitué des fautes logicielles de développement et des fautes naturelles en opération.

Nous avons également présenté dans ce chapitre une architecture générique pour les systèmes interactifs proposant des interactions de type WIMP qui permet d'identifier les composants logiciels sur lesquels nous pourrons appliquer notre approche pour concevoir et développer des systèmes interactifs

sûrs de fonctionnement. Cette architecture a ensuite été instanciée dans le cas des cockpits interactifs qui constituent le contexte applicatif de la thèse. Cette instanciation de l'architecture, nous permet d'appliquer notre approche sur des composants concrets.

Enfin, nous avons présenté le principe général de notre approche pour concevoir et développer des systèmes interactifs sûrs de fonctionnement. Cette approche repose sur une approche pour la conception et le développement de systèmes interactifs zéro-défaut et sur l'introduction des mécanismes de tolérance aux fautes dans les systèmes interactifs.

Les chapitres suivants présentent plus en détail les deux méthodes de l'approche proposée. Ainsi, nous présentons dans le Chapitre 5 une approche à base de modèle pour une conception et un développement zéro-défaut des systèmes interactifs au travers la modélisation des composants du système à l'aide d'une technique de description formelle nommée ICO. Nous présentons dans le Chapitre 6 une architecture logicielle tolérante aux fautes pour les systèmes interactifs. Enfin, nous présentons dans le Chapitre 7 des outils et des moyens pour la mise en œuvre de notre approche.

Chapitre 5. Approche à base de modèles pour des systèmes interactifs zéro-défaut

Sommaire

Au vu du modèle de faute que nous venons de présenter, nous proposons dans un premier temps une approche pour la prévention des fautes logicielles de développement. Dans ce but, ce chapitre présente notre approche à base de modèles pour la conception et le développement zéro-défaut de systèmes interactifs. Elle est fondée sur la description concise, complète et non ambigüe des différents composants du système interactif grâce à leur modélisation à l'aide d'une notation formelle appelée ICO.

La première section présente le principe de notre approche à base de modèles pour la conception et le développement de systèmes interactifs zéro-défaut. Pour cela, nous définissons le concept de zéro-défaut et décrivons notre approche pour nous en approcher autant que possible.

La deuxième section présente la notation formelle ICO, les différents éléments qui la composent et comment ceux-ci nous permettent de modéliser les aspects essentiels des systèmes interactifs.

La troisième section présente comment notre approche à base de modèles peut-être appliquée à l'architecture générique pour les systèmes interactifs que nous avons présentée en section 4.3.

La quatrième section illustre notre approche sur un exemple simple. Cette section nous permet d'illustrer la modélisation de tous les composants définis par notre architecture (voir section 4.3) et de mettre en avant les différents concepts et techniques utilisés.

5.1 Principe de l'approche proposée

5.1.1 Conception zéro-défaut

Un logiciel zéro-défaut est un logiciel ne contenant aucune faute de développement logiciel et qui se comporte exactement comme attendu. Concevoir et développer un logiciel zéro-défaut est

théoriquement possible mais très difficile (Hamilton 1986). Il est cependant possible de concevoir des logiciels s'approchant du zéro-défaut (c'est-à-dire avec un fort taux de probabilité de ne pas avoir de défauts). Pour cela, il est nécessaire d'utiliser des méthodes formelles permettant de prévenir et d'éliminer les défauts du logiciel. Ces méthodes peuvent être appliquées à plusieurs niveaux du processus de développement : au niveau des spécifications, de la vérification et de la validation ainsi que du test du logiciel (Stavely 1998).

Le premier pas pour obtenir un logiciel zéro-défaut est de mettre en place des méthodes de prévention des fautes qui pourront être par la suite complétées par des méthodes d'élimination telles que le test du logiciel (Stavely 1998). Nous nous intéressons dans cette thèse à cette première étape, c'est-à-dire, aux méthodes de prévention des fautes permettant de construire un logiciel contenant le moins de fautes possibles.

Nous avons proposé dans le chapitre précédent une architecture logicielle pour les systèmes interactifs. Cette architecture logicielle participe à l'effort de conception zéro-défaut du système en définissant les composants logiciels des systèmes interactifs mais n'est pas suffisante car elle ne permet pas de spécifier le comportement de l'ensemble du système (en particulier les liens et relations entre les composants). Il est important de pouvoir spécifier le comportement des différents composants logiciels du système afin de pouvoir le vérifier. Dans le domaine des systèmes critiques, les modèles formels sont particulièrement utilisés pour décrire le comportement du système car ils permettent de fournir un support à la vérification et à la validation du système (Storey 1996).

5.1.2 Modélisation formelle des systèmes interactifs

Dans cette thèse, nous nous intéressons plus particulièrement aux systèmes interactifs. Pour concevoir et développer des systèmes interactifs zéro-défaut, nous proposons de modéliser de manière concise, complète et non ambigüe leur comportement à l'aide d'une notation formelle. Il est important de noter que cette modélisation devra, dans un second temps, être complétée par des approches de vérification et de validation.

Le besoin de notation formelle pour la modélisation du comportement des systèmes interactifs a été exprimé et étudié dans de nombreux travaux tels que ceux de (Palanque et Bastide 1994), (Johnson 1995), (Thimbleby, User-Centered Methods Are Insufficient for Safety Critical Systems 2007) ou (A. J. Dix 1995). Ces travaux permettent de mettre en évidence les avantages de l'utilisation de notations formelles pour la description du comportement des systèmes en montrant qu'ils permettent un certain nombre d'avantages non spécifiques aux aspects interactifs du système :

- Décrire le comportement des systèmes de manière complète et non ambigüe.
- Pratiquer des activités d'analyse et de vérification sur les modèles et donc sur le comportement du système.
- Améliorer la communication entre les différents acteurs du cycle de développement.

Nous avons identifié en section 2.5.5 les besoins spécifiques aux systèmes interactifs pour sélectionner une notation pour la description comportementale des systèmes interactifs. Tout d'abord, celle-ci doit avoir un pouvoir d'expression permettant de décrire les aspects spécifiques aux systèmes interactifs que nous citons ci-dessous :

- La description des objets et de leur valeur.
- La description des états.
- La représentation des événements.
- La représentation des aspects temporels.
- La représentation des comportements concurrents et de l'instanciation dynamique.
- Description de la présentation et des activations qui font suite à une action de l'utilisateur.

En plus de ces besoins de pouvoir d'expression, nous nous intéressons à une notation formelle permettant de décrire tous les composants logiciels de notre architecture. Celle-ci doit également être

outillée afin de permettre la description de systèmes compliqués tels que les applications interactives dans les cockpits d'avions.

Toutes ces exigences nous amènent à choisir la notation ICO (Interactive Cooperative Objects) car c'est la seule qui répond aux besoins de conception et de développement des systèmes interactifs critiques (voir section 2.5.5). Nous décrivons cette notation dans la section suivante telle qu'elle a été décrite dans les travaux de (Hamon 2014) qui présentent sa version la plus actuelle.

5.2 ICO : une notation formelle pour la spécification des systèmes interactifs

Dans cette section, nous présentons la notation formelle que nous utilisons : ICO (pour Objets Coopératif Interactif). Il est important de noter que cette thèse n'apporte pas de contribution sur la notation ICO, nous avons cependant choisi de détailler cette notation dans ce chapitre afin de faciliter la lecture.

La notation ICO s'appuie sur les concepts de la programmation orientée-objet pour décrire les aspects structurels des systèmes interactifs et sur les concepts des réseaux de Petri haut-niveau (Genrich 1991) pour décrire les aspects comportementaux. Une première version de cette notation a été définie par les travaux de (Palanque 1992) et s'appuie sur l'utilisation des réseaux de Petri à objets comme présenté dans les travaux de (Bastide et Palanque 1990). Nous présentons dans ce document la dernière version de la notation telle qu'elle a été définie par les travaux de (Hamon 2014).

Pour mieux comprendre la notation ICO, nous présentons tout d'abord les réseaux de Petri sur lesquels elle s'appuie. Nous présentons ensuite la notation CO (pour Cooperative Objects ou Objets Coopératifs) dont la notation ICO est une extension. Enfin, nous présentons la notation ICO. Nous ne présentons ici que les éléments essentiels à la compréhension de la notation et à son utilisation dans le cadre de cette thèse. Une présentation plus complète de ces trois notations (réseaux de Petri, CO et ICO) peut-être trouvée dans les travaux de (Hamon 2014). Il est important de noter que les différentes illustrations présentées dans ce chapitre correspondent à celles fournies par l'outil PetShop qui permet notamment d'éditer les modèles ICO. Nous présentons cet outil en section 7.1.

5.2.1 Les réseaux de Petri et les réseaux de Petri haut-niveau

Les réseaux de Petri (Petri 1962) sont une technique de description formelle permettant de décrire le comportement dynamique des systèmes à événements discrets. Concrètement, ils sont un outil graphique s'appuyant sur un modèle mathématique qui permet de modéliser explicitement les notions d'état et de changement d'état en prenant en compte un nombre infini d'états.

Figure 5.1. Représentation des places dans les réseaux de Petri

Un réseau de Petri est un graphe comprenant deux types de nœuds (les places et les transitions) reliés par des arcs orientés. Les places sont représentées par des ellipses (voir Figure 5.1-a) et permettent de modéliser les différents états du système. Elles peuvent contenir des jetons qui sont représentés par des cercles violets marqués d'un chiffre représentant le nombre de jetons présents dans la place (voir Figure 5.1-b). Ceux-ci représentent les variables d'état du système. L'état d'un réseau de Petri à un instant donné (et donc du système associé) est défini par son marquage, c'est-à-dire par la distribution et la valeur des jetons dans ses places.

a) Transition franchissable *b) Transition non franchissable*

Figure 5.2. Représentations des transitions dans les réseaux de Petri

L'évolution de l'état d'un réseau de Petri est conditionnée par le franchissement des transitions. Les transitions sont représentées par des rectangles (voir Figure 5.2) et permettent de caractériser l'évolution des jetons en fonction du marquage du réseau et du parcours défini par les arcs.

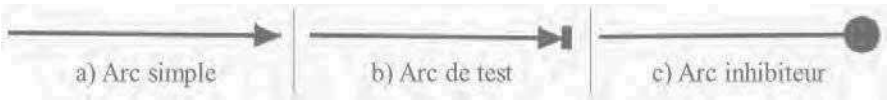


Figure 5.3. Représentation des différents types d'arcs dans les réseaux de Petri

Les arcs sont des arcs orientés (voir Figure 5.3) qui permettent de relier une place à une transition ou inversement. Ils permettent de définir le parcours des jetons en conditionnant la franchissabilité des transitions. Ainsi, une transition est franchissable si et seulement si le nombre de jetons dans ses places d'entré (places reliées par un arc entrant à la transition) est supérieur ou égal au poids des arcs simples (voir Figure 5.3-a) reliant respectivement chacune des ces places à cette transition. Une transition franchissable est représentée par un rectangle violet (voir Figure 5.2-a) alors qu'une transition non franchissable est représentée par un rectangle gris (voir Figure 5.2-b). Le franchissement d'une transition ne met en jeu que les jetons contenus dans les places d'entré et de sortie. Si une transition est franchissable, elle est immédiatement franchie.

Le franchissement d'une transition consomme un jeton présent dans la place d'entrée en le retirant et dépose un jeton dans la place en sortie. Ce comportement est illustré en Figure 5.4. Le nombre de jetons consommés dans la place d'entrée et déposés dans la place de sortie dépend du poids des arcs associés.

Figure 5.4. État d'un réseau de Petri avant (a) et après (b) un franchissement de transition

Les places peuvent être reliées aux transitions par des arcs spéciaux, ceux-ci modifient alors les conditions de franchissabilité des transitions. Nous retiendrons ainsi les arcs de test (voir Figure 5.3-b) et les arcs inhibiteurs (voir Figure 5.3-c).

Les arcs de test rendent une transition franchissable si et seulement si la place en entrée contient au moins un jeton. Celui-ci ne sera pas consommé par le franchissement de la transition. Ce comportement est illustré en Figure 5.5.

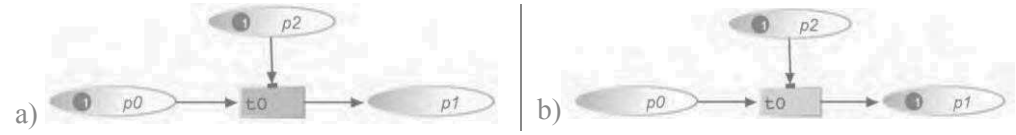


Figure 5.5. État d'un réseau de Petri avant (a) et après (b) le franchissement de transition conditionnée par un arc de test

Les arcs inhibiteurs rendent une transition franchissable si et seulement si la place en entrée ne contient pas de jeton. Ce comportement est illustré en Figure 5.6.

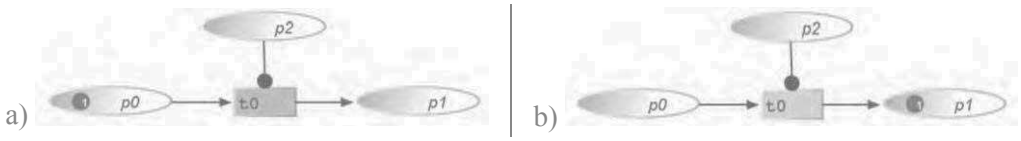


Figure 5.6. État d'un réseau de Petri avant (a) et après (b) le franchissement de transition conditionnée par un arc inhibiteur

Les réseaux de Petri temporisés (Ghezzi, et al. 1989) permettent de modéliser le temps de manière quantitative au travers l'utilisation de *transitions temporisées*. Une transition temporisée est franchie seulement après un certain temps après être devenue franchissable. Ce temps est défini en millisecondes et présenté entre crochets. Ce comportement est illustré en Figure 5.7 où l'on peut voir que la transition `t0_` n'est franchie qu'après 200ms.

Figure 5.7. État d'un réseau de Petri avant (a) et après (b) le franchissement d'une transition temporisée

Par construction, plusieurs transitions peuvent être en conflit en ayant les mêmes places et conditions d'entrées comme illustré en Figure 5.8. Dans ce cas, le réseau sera indéterministe : le choix de la transition franchie sera fait de manière aléatoire.

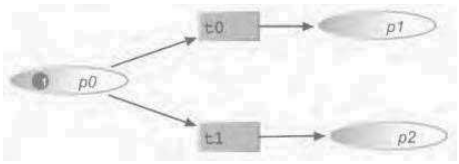


Figure 5.8. Exemple de comportement indéterministe dans les réseaux de Petri

Les réseaux de Petri à objet (ou OPN pour Object Petri Nets), définis par les travaux de (Jensen 1987), introduisent les caractéristiques suivantes :

- Les jetons peuvent être des références à des objets ;
- Les arcs sont étiquetés par des noms de variables permettant de décrire le flot des objets ;
- Les transitions peuvent contenir des actions qui sont effectuées lors du franchissement ;
- La franchissabilité d'une transition peut être conditionnée (en plus de la présence ou l'absence de jetons) par une précondition portant sur la valeur de ces jetons ;
- La franchissabilité d'une transition peut également être conditionnée par la propriété d'*unification*.

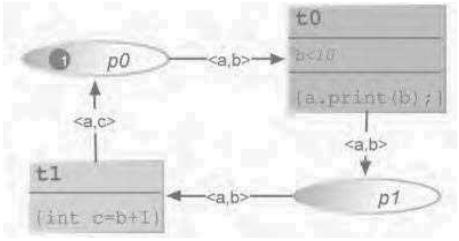


Figure 5.9. Exemple de réseau de Petri à objets

La Figure 5.9 présente un exemple de réseau de Petri à objets. Les jetons circulant dans le réseau sont des couples `(a,b)` (respectivement `(a,c)`) constitués d'un objet `a` ayant une méthode `print(int)` et d'un entier `b` (respectivement `c`). La transition `t0` possède une précondition sur la valeur de l'objet `b`, elle est franchissable si et seulement si la valeur de l'entier `b` contenu dans le jeton de la place `p0` est strictement inférieur à 10 (c'est le cas sur la figure) ; cette transition possède également une action `a.print(b)` qui exécute la méthode `print` de l'objet `a` avec comme paramètre l'entier `b`. La transition `t1` ne possède pas de précondition, mais possède une action : `int c = b +1`.

Supposons que dans l'état initial le jeton de la place `p0` contienne un objet `x` du bon type et d'un entier `0`, autrement dit, un couple `<x,0>`. Lors du franchissement de la transition `t0`, l'action `print(0)` est effectuée sur l'objet `x` et le jeton `<x,0>` est transmis à la place `p1`. La transition `t1` est alors franchissable. Lors de son franchissement, un jeton est transmis à la place `p0`, contenant les valeurs d'objets modifies suite à l'action effectuée lors du franchissement de `t1`, autrement dit `<x,1>`. La transition `t0` est alors de nouveau franchissable, tant que la valeur de `b` est inférieure à 10 (c'est-à-dire jusqu'à ce que la transition `t1` ait été franchie dix fois).

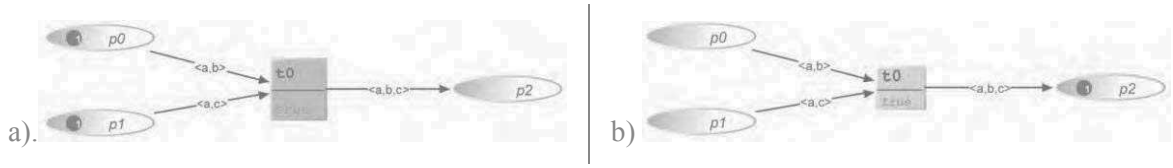


Figure 5.10. Exemple de franchissement dans un réseau de Petri à objets avant (a) et après (b) unification

La Figure 5.10 illustre la propriété d'unification. Lorsque les arcs d'entrée d'une transition sont étiquetés avec un nom de variable commun, la transition est franchissable si et seulement si les places d'entrée contiennent chacune un jeton ayant une valeur identique pour cette variable étiquetée. C'est ce

comportement qui est reporté en Figure 5.10 où la valeur de la variable `a` est égale pour le jeton contenu dans la place `p0` et pour celui contenu dans la place `p1` ; si les deux jetons n'avait pas contenu une valeur de variable `a` égale, la transition `t0` n'aurait pas été franchissable.

5.2.2 Les Objets Coopératif (CO)

Le formalisme des Objets Coopératifs est constitué d'un ensemble de classes (les CO-classes) qui définissent des objets coopératifs (les instances). Une CO-classe est constituée d'une interface Java décrivant les services qu'elle propose et d'un réseau de Petri à objet décrivant son comportement ; celui-ci est appelé ObCS (pour Structure de Contrôle de l'Objet ou Object Control Structure).

La communication entre les objets coopératifs est rendue possible par deux moyens : l'utilisation d'un protocole de communication de type client-serveur (en utilisant des appels des services définis par l'interface Java) ; ou l'utilisation d'une communication par événements.

Les services dans les objets coopératifs

L'interface logicielle Java de la CO-classe permet de définir les services offerts par la CO-classe. L'exécution de ces services est synchrone. Pour chacun de ces services, trois places sont générées dans l'ObCS (le réseau de Petri) correspondant à la CO-classe : une place d'entrée (`SIP`), une place de sortie (`SOP`) et une place d'exception (`SEP`). L'appel du service consiste à déposer un jeton contenant les paramètres de l'appel dans la place d'entrée du service (`SIP`). Rendre le service revient à déposer un jeton contenant le résultat de la requête dans la place de sortie du service (`SOP`). Enfin, la notification d'une erreur d'exécution du service revient à déposer un jeton décrivant l'erreur de la requête dans la place exception (`SEP`). La Figure 5.11 présente un exemple de CO-classe proposant une méthode `add(int a,int b)` réalisant l'addition de deux entiers `a` et `b`. Lors d'un appel de cette méthode, un jeton est déposé dans la place `SIP_add`. La transition `add` est alors franchie et réalise l'action `c = a + b`. Un jeton contenant la valeur de `c` est alors placé dans la place `SOP_add`, rendant ainsi le service.

a)

```
public interface adder {
    public int add(int a,int b);
}
```

b)

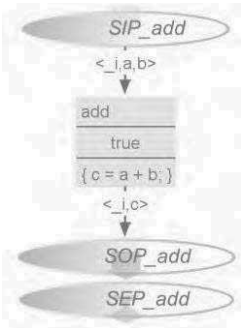


Figure 5.11. Exemple de CO-classe comprenant son interface Java (a) et son ObCS (b)

Les événements dans les objets coopératifs

En plus de la notion de services permettant la communication synchrone et unicast (un à un) entre objets coopératifs, la notion d'évènements a été intégrée dans les Objets Coopératifs afin de permettre la communication asynchrone et multicast (à plusieurs autres modèles). Cette communication est rendue possible grâce à des abonnements, une syntaxe pour le postage d'événements et un type de transitions permettant la réception d'un événement. La syntaxe d'abonnement à un événement est décrite en Figure 5.12-a ; la syntaxe d'envoi d'événement est décrite en Figure 5.12-b.

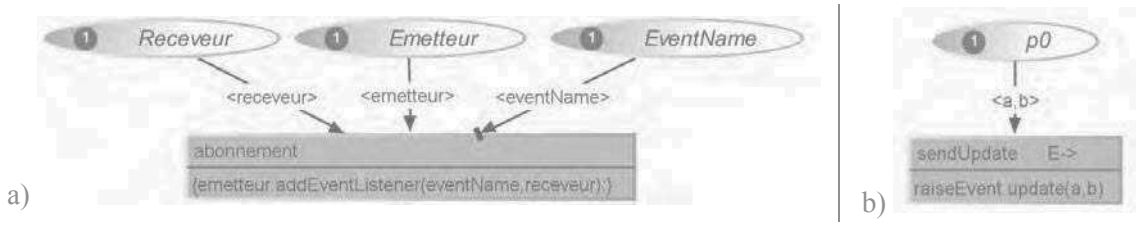


Figure 5.12. Abonnement à un événement (a) et envoi d'un événement (b)

La Figure 5.13 présente un exemple de transition réceptrice d'événement. Cette transition a été créée pour recevoir les événements `update` provenant de l'objet `emetteur`. Elle reçoit un ensemble de paramètres provenant de cet événement appelé `eventParams` (`a` et `b` dans cet exemple). Elle est conditionnée par deux préconditions : la première est du même type que les préconditions des transitions classiques (elle est réglée à `true` dans cet exemple) ; la seconde, appelée `eventCondition`, peut dépendre des paramètres transmis par l'événement (elle est également réglée à `true` dans cet exemple). Enfin, cette transition peut effectuer une action à la manière des transitions classiques (`{System.out.println("a::"+a+" b::"+b);}` dans cet exemple).

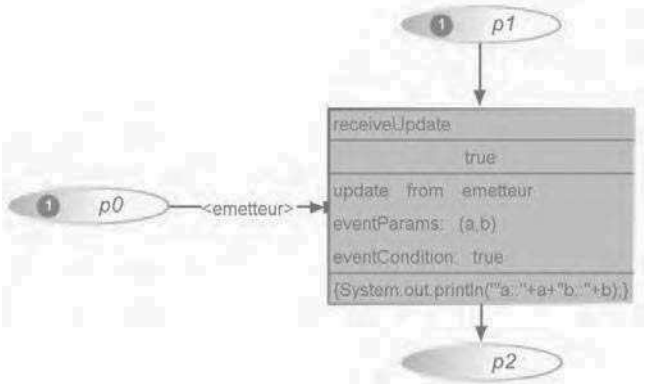


Figure 5.13. Exemple de transition réceptrice d'événements

La notation, associée à l'outil PetShop et son interpréteur (que nous présentons en section 7.1) permet également l'abonnement à des événements relatifs aux changements d'états d'un objet coopératif. Ces événements sont décrits dans le Tableau 5.1.

Nom	Événement
PlaceName_TokenAdded	Un jeton est entré dans la place PlaceName
PlaceName_TokenRemoved	Un jeton est sorti de la place PlaceName
PlaceName_MarkingReset	Le marquage de la place PlaceName a été réinitialisé
TransitionName_TransitionCompleted	La transition TransitionName a été franchie
EventName_Enabled	L'event handler de l'événement EventName a été activé
EventName_Disabled	L'event handler de l'événement EventName a été désactivé

Tableau 5.1. Événements correspondants aux changements d'états d'un objet coopératif

5.2.3 Les Objets Coopératifs Interactifs (ICO)

Les Objets Coopératifs Interactifs (ICO) sont une extension des CO (Objets Coopératifs). Cette extension permet de décrire les aspects interactifs des systèmes modélisés :

- Le comportement des différents composants du système sont décrits grâce à un ensemble de CO-Classes.
- Le lien avec l'apparence graphique du système interactif est décrit grâce à la partie présentation.

Le lien entre le comportement du système et l'évolution de son rendu graphique se fait grâce à deux fonctions : la fonction de rendu et la fonction d'activation. Elles permettent de maintenir la cohérence entre l'état de la CO-Classe et son apparence.

Dans un réseau de Petri, les places sont les variables d'état du système (l'état du système étant modélisé par une distribution de jetons dans les différentes places : le marquage) et les transitions sont les opérateurs de changement d'état. Les arcs, quant à eux, représentent les conditions pré-requises aux changements d'état et leurs effets sur les états du système. La fonction de rendu met en relation les places du comportement de l'application avec l'information affichée au moyen de trois types de méta-événements associées à trois cas d'évolution de l'état du système :

- L'entrée d'un jeton dans une place (événement `PlaceName_TokenAdded`).
- La sortie d'un jeton d'une place (événement `PlaceName_TokenRemoved`).
- La réinitialisation du marquage d'une place (événement `PlaceName_MarkingReset`).

La fonction d'activation a un double emploi. Elle permet premièrement de décrire le lien entre la disponibilité d'un service de l'application et la possibilité d'accomplir une action sur un objet de l'interface graphique. Deuxièmement, elle permet de décrire le lien entre les actions enregistrées sur l'interface graphique et le comportement de l'application. Elle utilise pour cela, deux types de métat-événements associés à deux cas d'évolution de l'état du système :

- La disponibilité nouvelle d'un service utilisateur (événement `EventName_Enabled`).
- L'indisponibilité nouvelle d'un service utilisateur (événement `EventName_Disabled`).

Les méta-événements sont des événements envoyés par l'interprète de l'outil PetShop (voir section 7.1) et qui représentent l'état du modèle.

5.2.4 Les ICompoNet

Les ICO permettent d'avoir une vue claire du comportement des composants interactifs. Cependant, il peut être intéressant, à la vue du nombre de modèles ICO pour une même application de bénéficier d'une vue de plus haut niveau permettant d'avoir un aperçu rapide des connexions entre les différents composants. C'est pour cette raison que les travaux de (Barboni 2006) ont proposé, associée à la notation ICO, l'utilisation d'un modèle de composant appelé ICompoNet. Chaque composant ICompoNet est constitué d'un modèle ICO permettant de décrire son comportement et d'un modèle CompoNet permettant de décrire les services et les événements fournis et reçus par le composant. Ce dernier s'appuie sur l'enveloppe de composant CCM (Corba Component Model) pour décrire les services et événements fournis et reçus par chaque composant ICompoNet.

Figure 5.14. Syntaxe graphique des CompoNet

La syntaxe graphique des CompoNet est présentée en Figure 5.14. Cette figure nous permet de mettre en évidence les quatre types de ports permettant de décrire les services et les événements fournis et reçus par le composant :

- *Les facettes* (`facet`) représentent l'ensemble des fonctionnalités fournies aux autres composants ;
- *Les réceptacles* (`receptacle`) représentent l'ensemble des fonctionnalités nécessaires au fonctionnement du composant ;
- *Les sources d'événements* (`eventSource`) représentent les événements produits par le composant ;
- *Les puits d'événements* (`eventSink`) représentent les événements que le composant est susceptible de recevoir.

La représentation CompoNet des composants ICompoNet permet de décrire les connexions entre les différents composants.

5.2.5 Synthèse

La présentation de cette notation met en évidence et confirme le fait qu'elle permette de modéliser tous les aspects essentiels à la description du comportement des systèmes interactifs que nous avons listé en section 5.1.2 :

- *La description des objets et de leur valeur* est supportée par le fondement de la notation ICO sur les objets de Petri à objets.
- *La description des états* est supportée par le fondement de la notation ICO sur les réseaux de Petri : un état correspond au marquage dans le réseau.
- *La représentation des événements* est supportée par les aspects spécifiques de la notation ICO et l'envoi et la réception d'événements.
- *La représentation des aspects temporels* est supportée par les aspects spécifiques de la notation ICO et les transitions temporisées.
- *La représentation des comportements concurrents et de l'instanciation dynamique* est supportée par le fondement de la notation ICO sur les réseaux de Petri et les différentes choses qu'ils permettent de modéliser.

Enfin, du fait de son fondement sur les réseaux de Petri, la notation ICO rend possible différents moyens d'analyse formelle des modèles ICO. Ces moyens permettent de supporter les activités de vérification et de validation du logiciel, nous les présentons en section 7.3.1.

5.3 Application de l'approche à notre architecture

La Figure 5.15 reprend l'architecture générique que nous avons proposée en section 4.3 en l'appliquant au domaine des cockpits interactifs : cette architecture respecte les contraintes imposées par le standard ARINC 661. Cette architecture spécifie également, du fait du domaine d'application des cockpits interactifs, les dispositifs d'entrée et sortie (KCCU et écran) ainsi que la constitution du noyau fonctionnel du système (un système avionique).

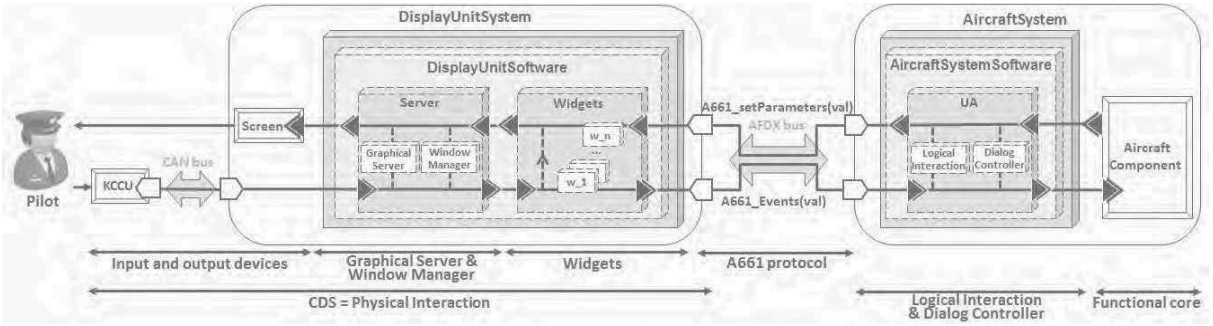


Figure 5.15. Architecture logicielle d'un système interactif de cockpit respectant le standard ARINC 661

L'application de notre approche vers une conception zéro-défaut à cette architecture correspond à la modélisation du comportement des différents composants de cette architecture à l'aide de la notation formelle ICO. Plus concrètement, il s'agit de modéliser le comportement du contrôleur de dialogue (`Dialog Controller`), de l'interaction logique (`Logical Interaction`), des widgets (`Widgets`) et du serveur (`Server`). Il est important de noter que cette approche a déjà été appliquée dans les travaux de (A. Tankeu-Choitat 2011) qui avaient montré son application pour la modélisation des widgets et du contrôleur de dialogue. Nous étendons ici cette modélisation à l'interaction logique et au serveur.

Nous proposons ainsi la modélisation des composants suivants :

- *Le contrôleur de dialogue* est représenté par un modèle unique qui définit le comportement de l'application. Il est très spécifique au noyau fonctionnel.
- *L'interaction logique* est décomposée en trois modèles correspondant à ces trois composants : la présentation, la fonction d'activation et la fonction de rendu.
- Le comportement de chaque type de *widget* est décrit par un modèle ICO.
- *Le serveur* est décomposé en deux parties fonctionnelles : la première permet de gérer l'initialisation de l'application et la seconde permet de gérer le rôle fonctionnel du serveur durant l'opération du système.

5.4 Illustration sur un exemple

Afin de mettre en évidence les techniques utilisées et les concepts permettant de modéliser chacun de ces composants, nous nous appuyons sur un exemple simple à travers lequel nous cherchons à illustrer les concepts suivants :

- La modélisation des aspects comportementaux de tous les composants logiciels de notre architecture :
  - Le serveur et ses différentes fonctionnalités ;
  - Les différentes classes de widgets (voir classification section 3.1.2.3) ;
  - L'interaction logique composée de la présentation et des fonctions d'activation et de rendu ;
  - Le contrôleur de dialogue.
- Les éléments de communication entre les différents composants logiciels.
- L'application de notre approche à base de modèles sur un système interactif respectant le standard ARINC 661 imposé par le contexte des cockpits d'avions civils.

La simplicité de l'exemple choisi permet de concentrer les efforts d'explications sur les concepts et techniques mis en œuvre plutôt que sur l'exemple en lui-même.

5.4.1 Présentation de l'exemple

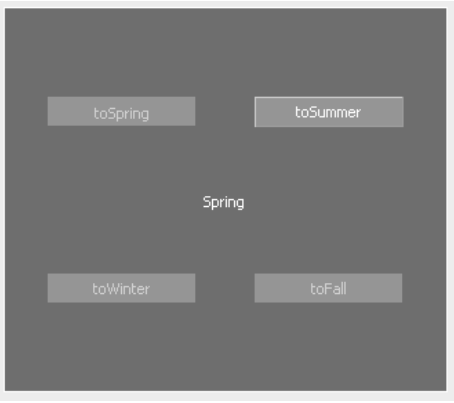


Figure 5.16. Capture d'écran de l'interface de l'application des quatre saisons

L'application « les 4 saisons », illustrée en Figure 5.16, est un très bon exemple illustratif. Celle-ci est composée en son centre d'un label permettant de notifier à l'utilisateur la saison actuelle (l'état de l'application). Ce label est associé à quatre boutons permettant à l'utilisateur de changer de saison (par exemple, le bouton `toSummer` permet à l'utilisateur de changer la saison actuelle et de la faire passer à `Summer`). Cette application doit respecter l'ordre logique des saisons : lorsque l'on est au printemps (`Spring`), on ne peut pas passer en hiver (`Winter`). Ainsi, qu'elle que soit la saison affichée, seul le bouton permettant de passer à la saison suivante doit être actif. C'est par exemple le cas dans l'état initial de l'application qui est illustré en Figure 5.16 où nous pouvons voir que la saison actuelle est le printemps (`Spring`) et que seul le bouton permettant de passer à l'été (`toSummer`) est actif.

Afin de respecter l'architecture logicielle imposée par le standard ARINC 661, cette application respecte l'organisation des widgets illustrée en Figure 5.17. Ainsi, l'élément de plus haut niveau est un *Layer*. Il permet de regrouper l'ensemble des widgets de l'application : un panel permettant de faire le regroupement graphique et fonctionnel des autres wigets ; les quatre boutons sont des *PicturePushButton* (`PPB`) présentant chacun un label permettant à l'utilisateur de connaître leur utilité ; enfin, le label permettant l'affichage de la saison en cours est un *Label*. Ces quatre types de widgets couvrent quatre classes des cinq classes de widgets que nous avons présenté en section 3.1.2.3 : le Panel est un widget de regroupement, le Label est un widget d'affichage et le PicturePushButton est un widget d'action.

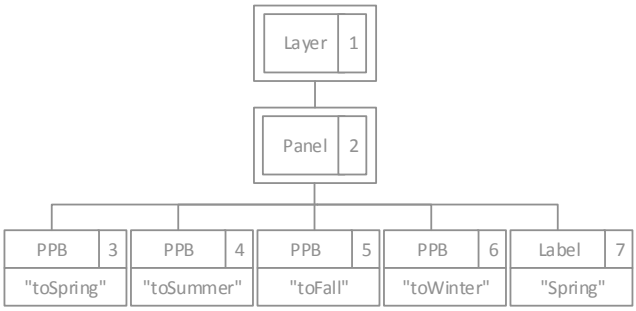


Figure 5.17. Arbre des widgets de l'application des quatre saisons

Nous présentons dans les sous-sections suivantes la modélisation des différents composants de ce système interactif.

5.4.2 Modélisation du contrôleur de dialogue

Le contrôleur de dialogue est responsable du comportement de l'interface, c'est ce modèle qui définit l'état du système (la saison actuelle) et les possibilités d'évolution, de changement d'état (l'enchaînement des saisons).

La Figure 5.18 présente le modèle ICO du contrôleur de dialogue de l'application. Ce modèle est composé de quatre places (`spring`, `summer`, `fall` et `winter`) qui représentent les quatre états possibles pour notre application. L'état de l'application est représenté par la présence d'un jeton dans la place correspondante à la saison en cours. Un changement d'état est représenté par le déplacement de ce jeton. Sur la Figure 5.18, le jeton est placé dans la place `spring`, cet état correspond à celui de l'interface présentée en Figure 5.16. Dans ce cas, le modèle du contrôleur de dialogue spécifie, du fait de la franchissabilité de la transition `t2`, qu'il est possible de passer dans l'état `summer`. Ce changement d'état est fait sur la réception de l'événement `toSummer` qui conditionne (avec la présence d'un jeton dans la place spring) le franchissement de la transition.

Les événements conditionnant les franchissements des quatre transitions de ce modèle sont envoyés par la fonction d'activation. Celle-ci est également responsable d'observer l'activation des event handler (qui correspond dans ce cas à la franchissabilité des transitions).

Les changements d'états du contrôleur de dialogue sont observés par la fonction de rendu.

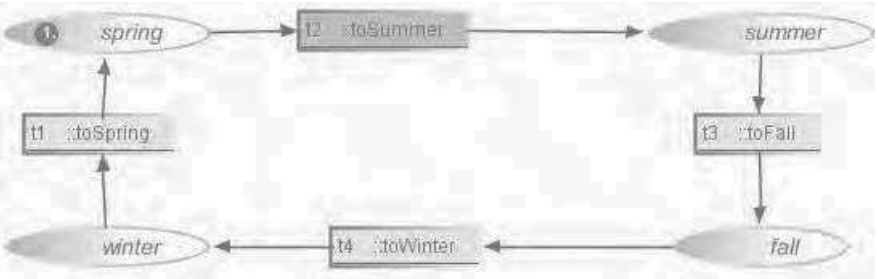


Figure 5.18. Modèle ICO du contrôleur de dialogue de l'application « Les quatre saisons »

5.4.3 Modélisation de l'interaction logique

5.4.3.1 La fonction d'activation

La fonction d'activation permet dans un premier temps d'envoyer des événements au contrôleur de dialogue (en fonction des actions de l'utilisateur) afin de permettre à celui-ci d'effectuer des changements d'états. Ce comportement est traduit à l'aide du Tableau 5.2.

Dans un second temps, elle permet de faire le lien entre le contrôleur de dialogue et la disponibilité d'un service de l'interface graphique, autrement dit, la possibilité pour l'utilisateur d'accomplir une action sur un widget. Ce comportement est traduit à l'aide du Tableau 5.3.

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_Spring	Clic	toSpring	{t1}
PPB_Summer	Clic	toSummer	{t2}
PPB_Fall	Clic	toFall	{t3}
PPB_Winter	Clic	toWinter	{t4}

Tableau 5.2. Spécification informelle de la fonction d'activation de l'application « Les 4 saisons » - Lien entre les actions de l'utilisateur et le contrôleur de dialogue

Objet d'interaction	Event handler	Événements	Méthode d'activation
PPB_Spring	toSpring	Enabled ou Disabled	setToSpringEnabled(b)
PPB_Summer	toSummer	Enabled ou Disabled	setToSummerEnabled(b)
PPB_Fall	toFall	Enabled ou Disabled	setToFallEnabled(b)
PPB_Winter	toWinter	Enabled ou Disabled	setToWinterEnabled(b)

Tableau 5.3. Spécification informelle du rendu d'activation l'application « Les 4 saisons » - Lien entre le contrôleur de dialogue et la disponibilité des services

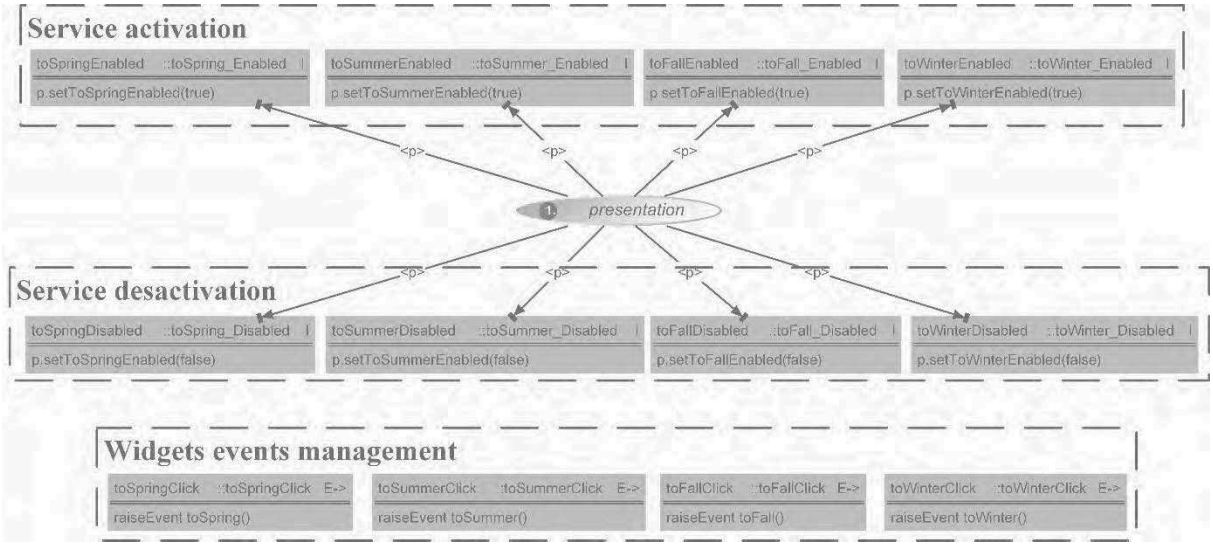


Figure 5.19. Modèle ICO de la fonction d'activation de l'application « Les 4 saisons »

La Figure 5.19 présente le modèle ICO de la fonction d'activation de notre application. Celle-ci est divisée en trois parties, les deux premières parties correspondent à la traduction du Tableau 5.3, la troisième partie correspond à la traduction du Tableau 5.2 :

- *Activation des services* : quand un event handler est activé, le service correspondant est activé.

  ***Exemple** : lorsque la transition* `t1` *du contrôleur de dialogue (voir Figure 5.18) devient franchissable (arrivée d'un jeton dans la place* `winter`*), la fonction d'activation reçoit un événement* `toSpring_Enabled` *correspondant à la notification de l'activation de l'event handler* `toSpring`*. Dans ce cas, la transition* `toSpringEnabled` *est franchie et appelle l'activation du service* `toSpring` *sur la présentation. Cet appel de service sera traduit (par la présentation) par l'activation du PPB_Spring.*

- *Désactivation des services* : quand un event handler est désactivé, le service correspondant est désactivé.

  ***Exemple** : lorsque la transition* `t1` *du contrôleur de dialogue (voir Figure 5.18) est franchie, la fonction d'activation reçoit un événement* `toSpring_Disabled` *correspondant à la notification de la désactivation de l'event handler* `toSpring`*. Dans ce cas, la transition* `toSpringDisabled` *est franchie et appelle la désactivation du service* `toSpring` *sur la présentation. Cet appel de service sera traduit (par la présentation) par la désactivation du PPB_Spring.*

- *Gestion des événements provenant des actions utilisateurs* : transformation des événements provenant des actions utilisateurs en événements compréhensibles par le contrôleur de dialogue.

  ***Exemple** : si l'application est dans l'état* `spring`*, le PicturePushButton* `toSummer` *est activé. Si l'utilisateur clique sur ce bouton, la fonction d'activation reçoit l'événement* `toSpringClick` *(provenant de la présentation) et elle notifie le contrôleur de dialogue de cette action de l'utilisateur en lui envoyant l'événement* `toSpring`*. Ce comportement est réalisé par le franchissement de la transition* `toSpringClick`*.*

5.4.3.2 La fonction de rendu

La fonction de rendu permet de modifier le rendu de l'application en fonction des changements d'états du dialogue. Elle est représentée par le Tableau 5.4. La Figure 5.20 présente le modèle ICO correspondant au comportement de la fonction de rendu. Chaque ligne du Tableau 5.4 est décrite par une transition.

État	Place	Événement(s)	Méthode de rendu
Spring	spring	Jeton entré Marquage réinitialisé	displayText("Spring")
Summer	summer	Jeton entré	displayText("Summer")
Fall	fall	Jeton entré	displayText("Fall")
Winter	winter	Jeton entré	displayText("Winter")

Tableau 5.4. Spécification informelle de la fonction de rendu de l'application « Les 4 saisons »

***Exemple** : lorsqu'un jeton est déposé dans la place* `spring` *du contrôleur de dialogue (lorsque l'on passe dans l'état* `spring`*), la fonction de rendu reçoit l'événement* `spring_TokenAdded`*, ce qui va avoir pour conséquence le franchissement de la transition* `springTokenAdded` *et l'appel de la méthode* `displayText("Spring")` *sur la présentation qui modifiera alors l'affichage du texte du Label en* `Spring`*.*

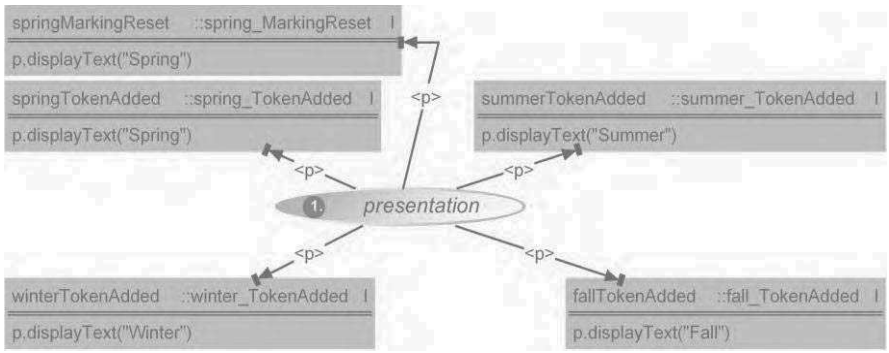


Figure 5.20. Modèle ICO de la fonction de rendu de l'application « Les 4 saisons »

5.4.3.3 La présentation

La présentation a pour but de faire la liaison entre les widgets et les fonctions d'activation et de rendu. Son modèle ICO est présenté en Figure 5.21. La présentation propose deux types de méthodes :

- Des méthodes permettant l'activation et la désactivation des services utilisateur de l'interface. Dans le cas de notre application, il s'agit des méthodes `setToSpringEnabled`, `setToSummerEnabled`, `setToFallEnabled`, `setToWinterEnabled` qui appelle l'activation et la désactivation des *PicturePushButton* correspondants (il est important de noter que l'activation et la désactivation d'un *PicturePushButton* entraîne la modification de son rendu graphique : s'il est désactivé, il est présenté grisé à l'utilisateur) ;

- Des méthodes permettant la modification du rendu graphique de l'interface. Dans le cas de notre application, il s'agit de la méthode `displayText` qui appelle la modification du texte affiché par le *Label* ;

La présentation est également responsable de transformer les événements provenant des widgets (événements faisant suite aux actions utilisateurs) à la fonction d'activation. Dans le cas de notre application, cette partie est gérée par les transitions `triggerToSpringClick`, `triggerToSummerClick`, `triggerToFallClick` et `triggerToWinterClick` qui sont franchies sur la réception des événements `A661_EVT_SELECTION` envoyés par les *PicturePushButton* (lorsqu'ils reçoivent un clic provenant d'une action utilisateur alors qu'ils sont actifs) et qui envoient un événement associé (par exemple `toSpringClick` lorsqu'il s'agit du PPPB_Spring) à la fonction d'activation.

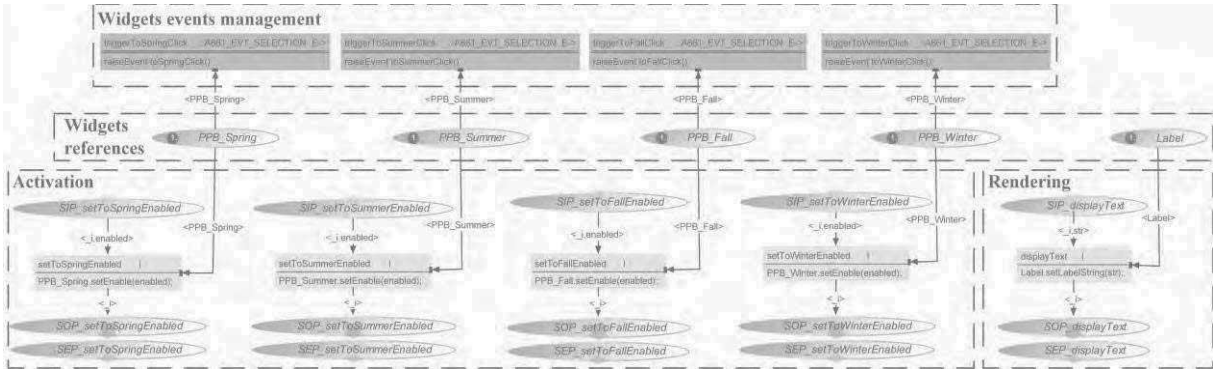


Figure 5.21. Modèle de la présentation pour l'application « Les quatre saisons »

5.4.4 Modélisation des widgets

5.4.4.1 Généralités

De façon générale, le comportement des widgets n'est pas décrit dans le standard ARINC 661 (voir section 3.1.2.3) même s'il peut être parfois brièvement défini dans la description textuelle du widget. Le comportement du widget, tel que défini par le standard ARINC 661, se résume à la définition de ses paramètres et des événements qu'il peut envoyer.

Comme expliqué dans les travaux de (Barboni 2006) et (A. Tankeu-Choitat 2011), pour obtenir le modèle ICO d'un widget, il est nécessaire de déterminer trois groupes d'éléments :

- Ses paramètres (modifiables en cours d'exécution ou non) ;
- Les événements que celui-ci envoie ;
- Si celui-ci est interactif ; autrement dit, s'il offre à l'utilisateur la possibilité d'interagir avec lui ou non.

Ces éléments permettent de définir le modèle CompoNet du widget. Ils sont mis en évidence par les différents ports du modèle CompoNet générique d'un widget présenté en Figure 5.22 :

- *setRuntimeModifiableParameter* : ce port regroupe les services proposés par le widget pour la modification de ses paramètres modifiables en cours d'exécution.
- *A661_EVENT* : ce port regroupe les envois d'événements du widget.
- *processInputDeviceEvent* : ce port regroupe les services spécifiques aux widgets interactifs, ces services correspondent au traitement des événements provenant des actions utilisateurs tel que la réception d'un clic.
- *setServerParameter* : ce port est également spécifique aux widgets interactifs et regroupe la modification de paramètres de rendu spécifiques aux widgets interactifs (tels que le paramètre `highlighted` qui met le widget en évidence lorsque celui-ci est survolé par le curseur).

Figure 5.22. Modèle CompoNet générique d'un widget

Pour chaque widget, le modèle CompoNet permet de déduire l'interface de son composant ICO et le squelette du réseau de Petri lui correspondant. Pour rappel, ce squelette est composé, pour chaque service proposé par le widget, des trois places `SIP`, `SEP` et `SOP` correspondantes au service (voir section 5.2.2). Une fois ce squelette réalisé, il s'agit alors de modéliser le comportement du widget. L'état de celui-ci est représenté par la distribution et la valeur des jetons dans les places du modèle. Les transitions permettent de modéliser la modification de l'état du widget. Enfin, la fonction de rendu d'un widget est réalisée par un modèle observant tous les changements d'états du widgets correspondants à une modification de l'affichage du widget (par exemple, la valeur du paramètre `Visible` du widget qui représente le fait que le widget soit dessiné ou non à l'écran). Cette responsabilité relève du serveur et est gérée par le graphe de scène.

L'application « les 4 saisons » comporte 4 types des widgets différents : le *Layer*, le *Panel*, le *PicturePushButton* et le *Label*. Le Tableau 5.5 regroupe les différentes caractéristiques de ces widgets. Il présente ainsi pour chaque widget, si celui-ci est interactif (colonne 2), les événements qu'il peut envoyer (colonne 3), ses paramètres non-modifiables en cours d'exécution (colonne 4) et ses paramètres modifiables en cours d'exécution (colonne 5). Au vu de la simplicité du comportement du layer (c'est un container ne possédant que deux paramètres représentant sa visibilité et son activation), nous ne présentons pas son modèle et nous ne le présentons par conséquent pas dans ce tableau.

Widget	Interactive	A661_Event	Parameter	
			DesignTime	Runtime
Panel	No		WidgetType WidgetID ParentID MotionAllowed	Visible Enable StyleSet PosX PosY SizeX SizeY
Label	No		WidgetType WidgetID ParentID SizeX SizeY MaxStringLength MotionAllowed Alignment	Visible StyleSet PosX PosY LabelSTring RotationAngle Font ColorIndex
PicturePushButton	Yes	A661_EVT_SELECTION	WidgetType WidgetID ParentID PosX PosY SizeX SizeY MaxStringLength Alignment PicturePosition	Visible Enable StyleSet LabelString PictureReference

Tableau 5.5. Caractéristiques des widgets de l'application

Ce tableau nous permet de définir les modèles CompoNet de nos trois widgets. Ceux-ci sont présentés en Figure 5.23

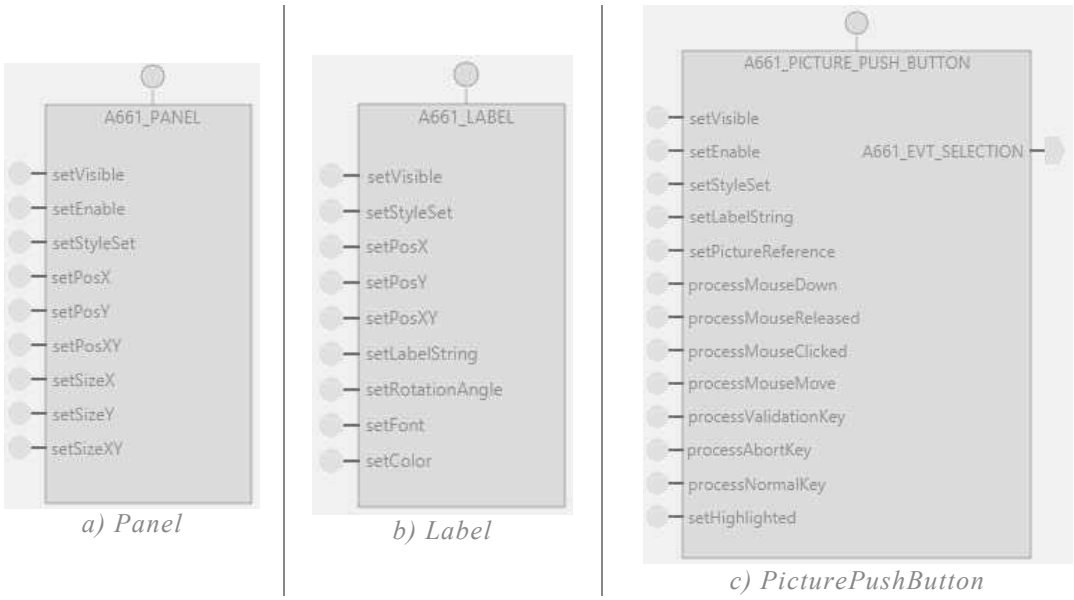


Figure 5.23. Modèles CompoNet des widgets de l'application

Le standard ARINC 661 ne décrit pas le « look & feel » des widgets, c'est-à-dire qu'il ne décrit ni le rendu graphique (look) ni le comportement des widgets et la manière dont l'utilisateur peut interagir avec eux (feel) (voir section 3.1.2.3). Ceux-ci sont généralement décrits par des standards internes aux avionneurs et à leurs fournisseurs. Chez Airbus, ce document précise notamment des traitements d'exceptions en cas d'erreur lors de la modification d'un paramètre en cours d'exécution. Le Tableau 5.6 présente la spécification de ces erreurs pour les widgets de notre application.

Widget	Error	Parameter	Error identifier
Panel			
Label	The value is out the range of the possible values associated to one application	ColorIndex	A661_OUT_OF_RANGE
	The value is out the range of the possible values associated to one application	Font	A661_OUT_OF_RANGE
	The LabelString length is higher than the specified MaxStringLength	LabelString	A661_ERROR_STRING_LENGTH
PicturePushButton	The value is out the range of the possible values associated to one application	PictureReference	A661_OUT_OF_RANGE
	The LabelString length is higher than the specified MaxStringLength	LabelString	A661_ERROR_STRING_LENGTH

Tableau 5.6. Description des erreurs pour les widgets de l'application

Les sous sections suivantes présentent la modélisation des différents widgets de notre application.

5.4.4.2 Modélisation d'un Panel

Un *Panel* est un widget permettant de regrouper plusieurs widgets dans un même cadre graphique. Ce n'est pas un widget interactif et il ne peut envoyer aucun événement. Son comportement se résume donc à ses actions lors de la demande de modification en cours d'exécution d'un de ces paramètres. Son modèle ICO complet est présenté en Figure 5.24. Nous expliquons ci-après plus en détail ses différentes fonctionnalités.

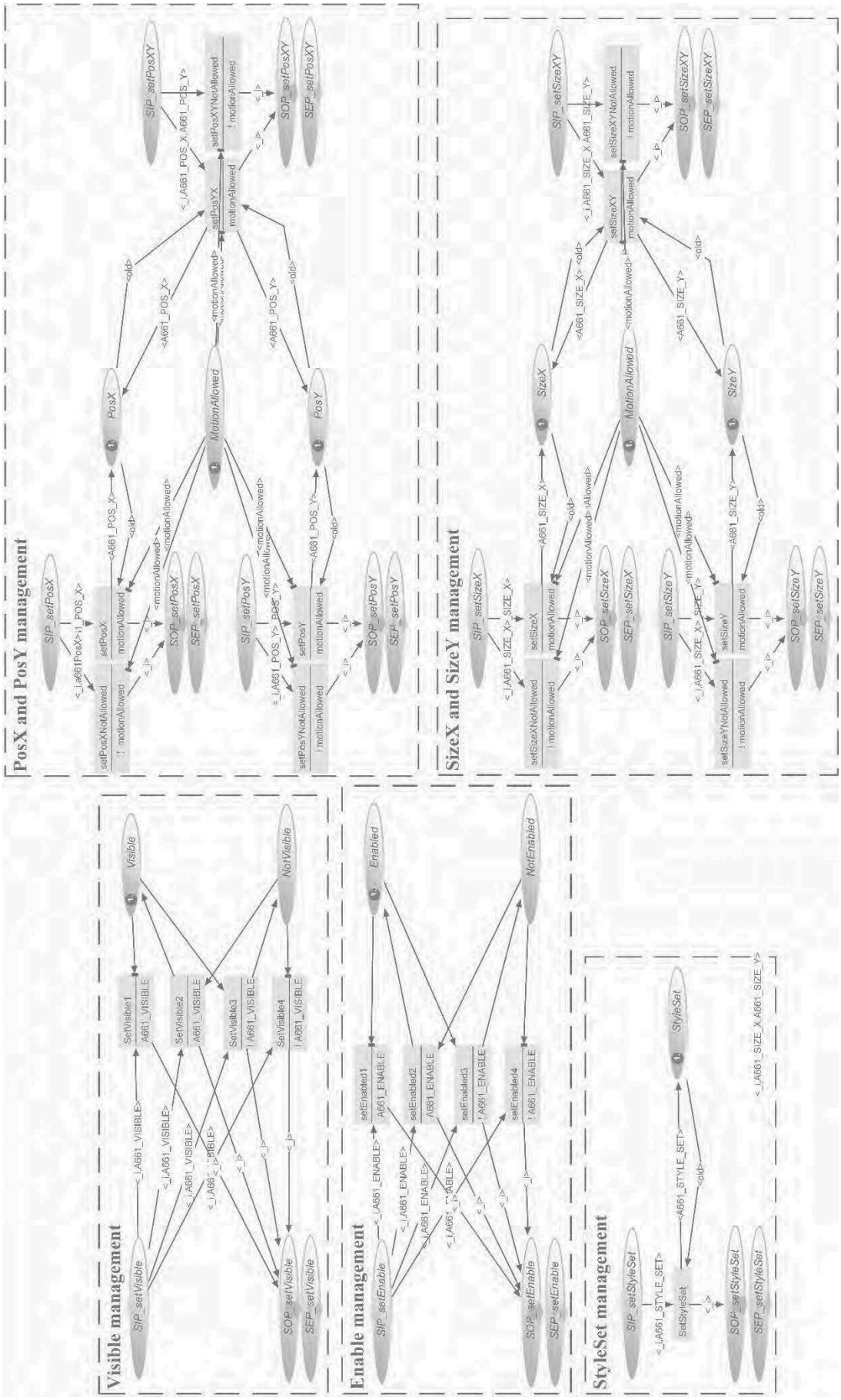
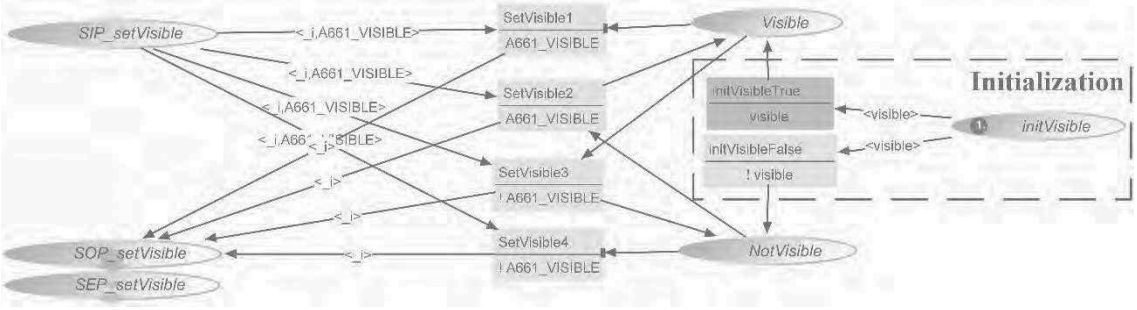


Figure 5.24. Modèle ICO d'un Panel

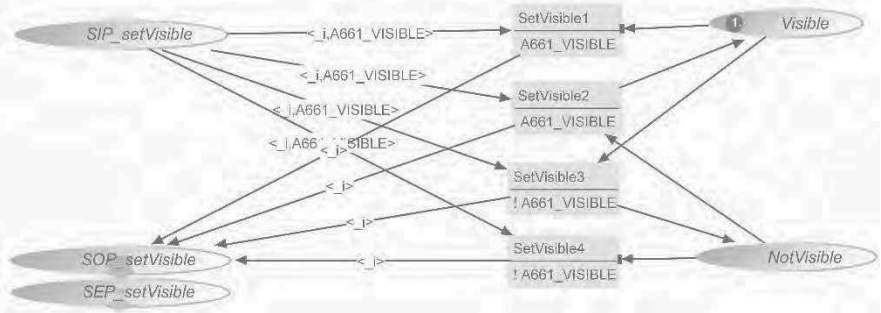
Le comportement de modification en cours d'exécution des paramètres `Visible` et `Enable` est le même. Nous le détaillons, pour le paramètre `Visible`, en Figure 5.25. La Figure 5.25-a montre le comportement spécifique à l'initialisation de ce paramètre. La valeur booléenne du paramètre `Visible` est stockée à l'initialisation dans la place `initVisible`. La transition `initVisibleTrue` (respectivement `initVisibleFalse`) permet d'initialiser l'état du modèle en plaçant un jeton dans la place `Visible` (respectivement `NotVisible`) si la valeur (`visible`) du jeton dans la place `initVisible` est `true` (respectivement `false`). Afin de faciliter la lecture du modèle, ce comportement d'initialisation est généralement caché comme on peut le voir sur la Figure 5.25-b qui présente la gestion du paramètre `Visible` après initialisation. C'est sous cette forme que nous présentons les modèles complets des widgets comme celui du *Panel* présenté en Figure 5.24.

Une fois initialisé, le modèle du *Panel* peut recevoir des appels de méthodes demandant la modification de la valeur du paramètre `Visible`. Un appel de méthode correspond au dépôt d'un jeton dans la place `SIP_setVisible`, ce jeton contient alors la nouvelle valeur désirée pour le paramètre `Visible`. Ce jeton est traité par le franchissement de l'une des quatre transitions `SetVisible` :

- Si un jeton est actuellement présent dans la place `Visible` (respectivement `NotVisible`), et que l'on reçoit un appel de méthode avec une nouvelle valeur `A661_VISIBLE=true` (respectivement `A661_VISIBLE=false`), la transition `SetVisible1` (respectivement `SetVisible4`) est franchie. Ce franchissement n'implique aucun changement d'état et permet de rendre le service par le dépôt d'un jeton dans la place `SOP_setVisible`.
- Si un jeton est actuellement présent dans la place `Visible` (respectivement `NotVisible`), et que l'on reçoit un appel de méthode avec une nouvelle valeur `A661_VISIBLE=false` (respectivement `A661_VISIBLE=true`), la transition `SetVisible2` (respectivement `SetVisible3`) est franchie. Ce franchissement permet la modification de la valeur du paramètre en déplaçant le jeton de la place `Visible` (respectivement `NotVisible`) vers la place `NotVisible` (respectivement `Visible`) et permet de rendre le service par le dépôt d'un jeton dans la place `SOP_setVisible`.



a) Partie du modèle correspondant à l'initialisation visible



b) Partie du modèle correspondant à l'initialisation cachée

Figure 5.25. Modèle ICO de la gestion du paramètre Visible

La Figure 5.26 présente la gestion de la modification du paramètre `StyleSet`. Ce paramètre ne nécessite pas de vérification particulière, ce qui implique que lors de la réception d'un appel de méthode pour la modification de ce paramètre, le jeton placé dans la place `SIP_setStyleSet` (contenant la nouvelle valeur désirée représentée par la variable `A661_STYLE_SET`) provoquera le franchissement de la

transition `SetStyleSet`. Celui-ci remplace le jeton contenu dans la place `StyleSet` par un nouveau jeton avec la valeur `A661_STYLE_SET` et rend le service en replaçant un jeton dans la place `SOP_setStyleSet`.

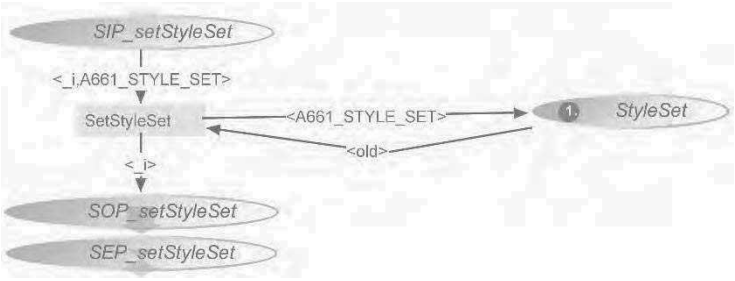


Figure 5.26. Modèle ICO de la gestion du paramètre StyleSet

Le comportement de modification en cours d'exécution des paramètres `PosX` et `PosY` et celui des paramètres `SizeX` et `SizeY` est similaire. Nous le détaillons, pour les paramètres `PosX` et `PosY`, en Figure 5.27. La modification des paramètres `PosX` et `PosY` est possible si et seulement si le paramètre `MotionAllowed` a la valeur `true`. Cette modification peut être faite de plusieurs manières : le widget propose soit la modification des paramètres `PosX` et `PosY` (méthodes `setPosX` et `setPosY`) soit la modification de ces deux paramètres grâce à l'appel d'une seule méthode (méthode `setPosXY`).

Un appel de la méthode `setPosX` (respectivement `setPosY`) résulte en la réception d'un jeton contenant la nouvelle valeur du paramètre dans la place `SIP_setPosX` (respectivement `SIP_setPosY`). Suite à la réception de ce jeton, deux scénarios sont possibles :

- La valeur du paramètre `MotionAllowed` est `true` : dans ce cas la transtion `setPosX` (respectivement `setPosY`) est franchissable car sa précondition est validée. Son franchissement permet le remplacement du jeton contenu dans la place `PosX` (respectivement `PosY`) par un jeton contenant la nouvelle valeur du paramètre.
- La valeur du paramètre `MotionAllowed` est `false` : dans ce cas, la transition `setPosXNotAllowed` (respectivement `setPosYNotAllowed`) est franchissable car sa précondition est validée. Son franchissement n'a aucun effet sur l'état du widget.

Le principe de la gestion d'un appel de la méthode `setPosXY` est exactement le même que celui que nous venons de décrire à l'exception près que celui-ci implique la modification simultanée des paramètres `PosX` et `PosY` (si la valeur du paramètre `MotionAllowed` est `true`).

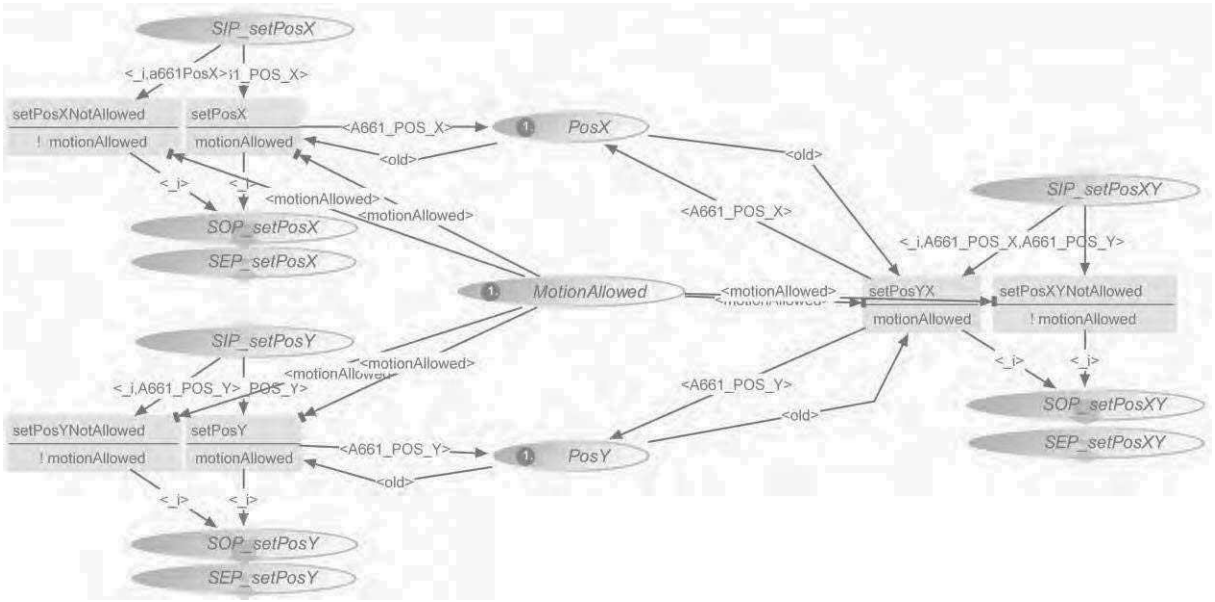


Figure 5.27. Modèle ICO de la gestion des paramètres PosX et PosY

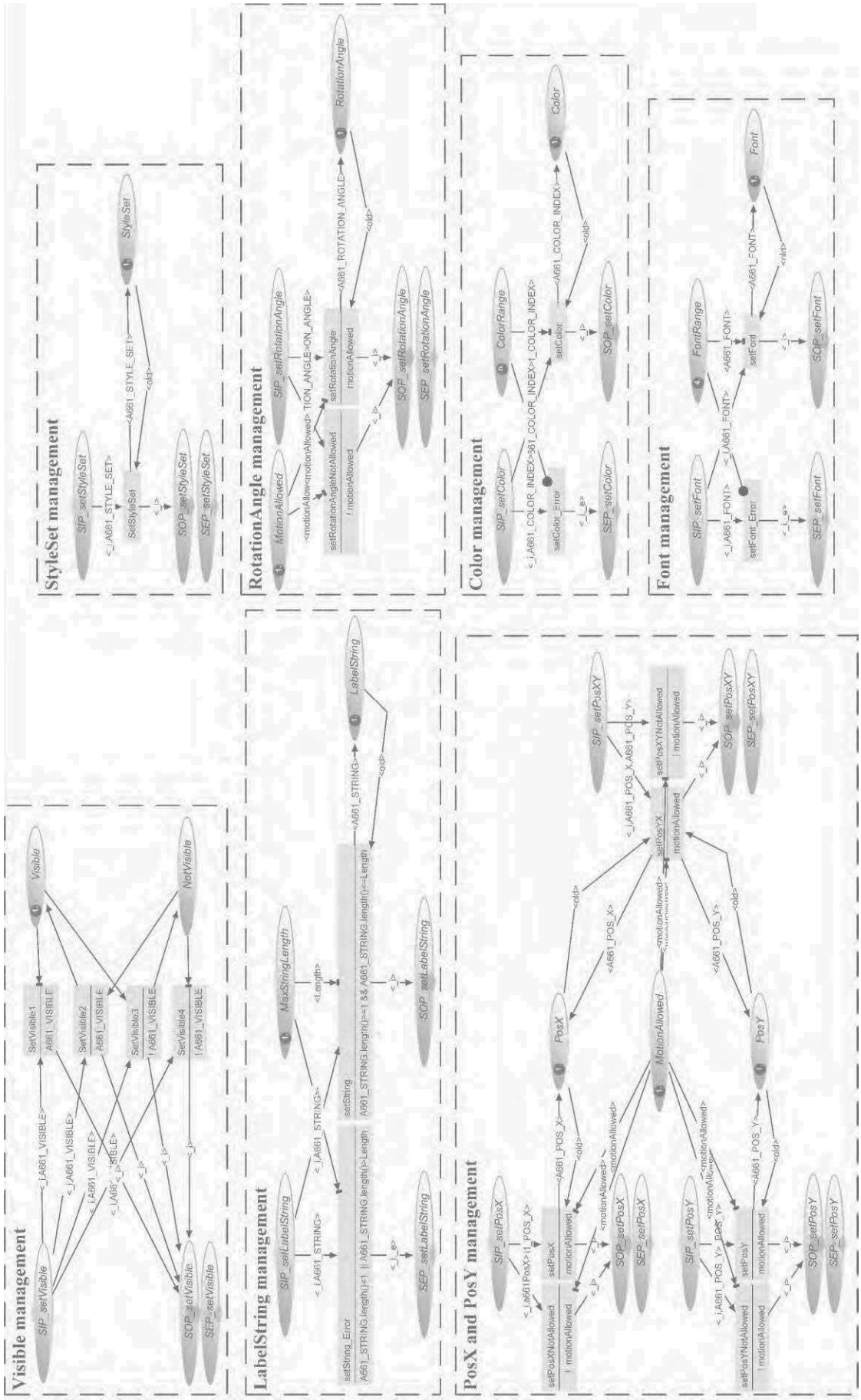


Figure 5.28. Modèle ICO d'un Label

5.4.4.3 Modélisation d'un Label

La Figure 5.28 présente le modèle ICO complet d'un *Label*. Celui-ci comporte un certain nombre de paramètres similaires à ceux du *Panel* : `Visible`, `PosX` et `PosY`, `StyleSet`. La modification de ces paramètres se fait de la même manière que pour le *Panel*.

Le paramètre `RotationAngle` est un paramètre permettant le mouvement du widget de la même manière que les paramètres `PosX` et `PosY`. Sa modification est donc dépendante, de la même manière que les paramètres `PosX` et `PosY`, de la valeur du paramètre `MotionAllowed`. Nous ne le détaillons donc pas plus ici.

Le comportement du widget lors de la modification en cours d'exécution du paramètre `LabelString` est présenté en Figure 5.29. La modification de ce paramètre nécessite une vérification : si le texte contenu dans sa nouvelle valeur est plus long que la longueur maximale autorisée (définie par le paramètre `MaxStringLength`), la modification ne doit pas se faire une erreur doit être levée. Ainsi, lors d'un appel de la méthode `setLabelString`, deux cas de figures sont possibles :

- La nouvelle valeur du texte a une longueur non nulle et inférieure à la longueur maximale : la transition `setString` est franchissable car sa précondition est validée. Son franchissement permet le changement de valeur du jeton contenu dans la place `LabelString`.
- La nouvelle valeur du texte a une longueur nulle ou supérieure à la longueur maximale : la transition `setString_Error` sera franchissable car sa précondition sera validée. Son franchissement n'influe pas la valeur du paramètre `LabelString` et envoie un message d'erreur : le service est rendu à travers le dépôt d'un jeton dans la place `SEP_setLabelString`.

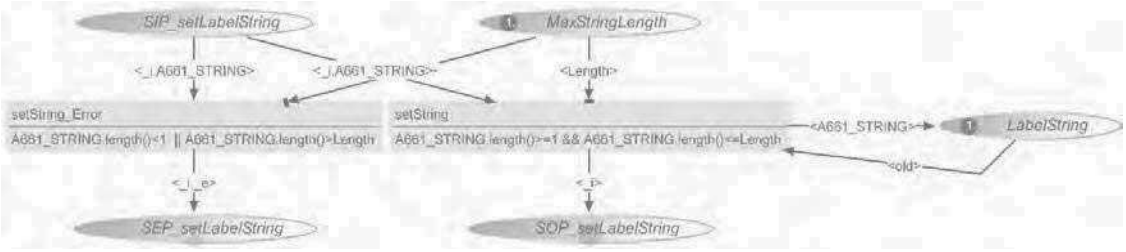


Figure 5.29. Modèle ICO de la gestion du paramètre LabelString

Le comportement du widget lors de la modification en cours d'exécution des paramètres `ColorIndex` et `Font` est similaire. Nous le détaillons, pour le paramètre `ColorIndex`, en Figure 5.30. La modification de ce paramètre nécessite également une vérification : avant d'effectuer la modification, le widget doit s'assurer que la nouvelle valeur qu'il a reçue est incluse dans l'ensemble des valeurs autorisées ; si c'est le cas, il effectue le changement, autrement il envoie un message d'erreur. Ainsi, lors d'un appel de la méthode `setColor`, deux cas de figure sont possibles :

- La nouvelle valeur du paramètre est incluse dans l'ensemble des valeurs autorisées : la transition `setColor` est franchissable car il existe un jeton dans la place `ColorRange` contenant la même valeur `A661_COLOR`. Son franchissement permet le changement de la valeur du jeton contenu dans la place `Color`.
- La nouvelle valeur du paramètre n'est pas incluse dans l'ensemble des valeurs autorisées : la transition `setColor_Error` est franchissable car il n'existe pas de jeton contenant la même valeur dans la place `ColorRange`. Son franchissement n'influe pas la valeur du paramètre `LabelString` et envoie un message d'erreur.

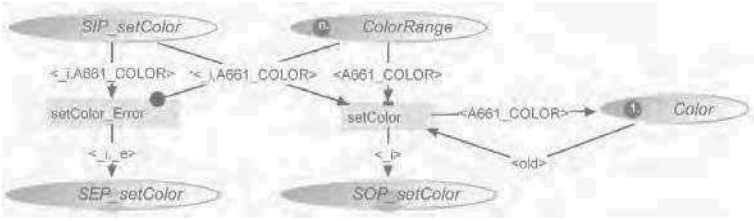


Figure 5.30. Modèle ICO de la gestion du paramètre Color

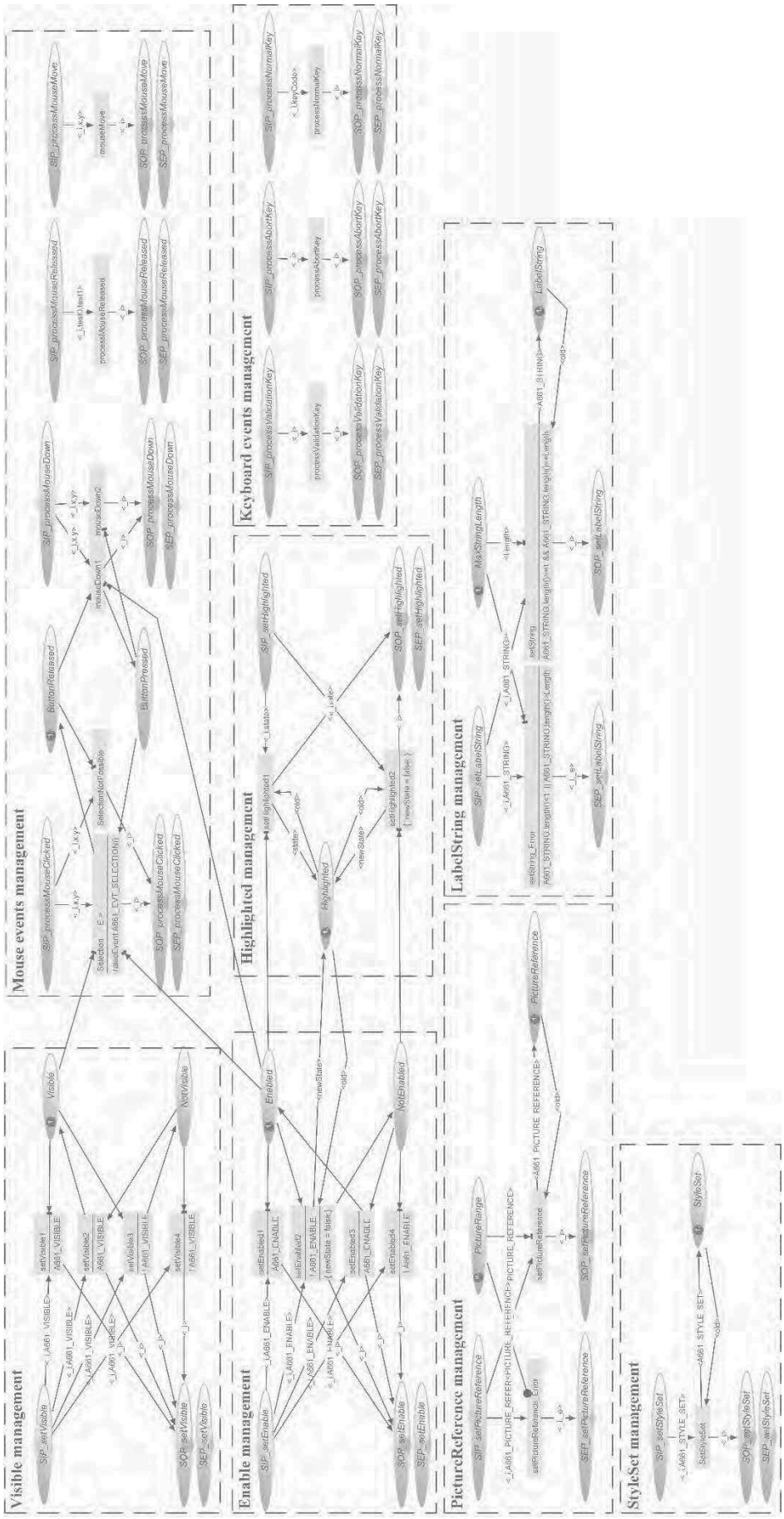


Figure 5.31. Modèle ICO d'un PicturePushButton

5.4.4.4 Modélisation d'un PicturePushButton

La Figure 5.31 présente le modèle ICO complet d'un *PicturePushButton*. Nous avons déjà détaillé pour le *Panel* et le *Label* le comportement lors de la modification en cours d'exécution des paramètres `Visible`, `StyleSet` et `LabelString`. La modification du paramètre `PictureReference` est similaire à celle du paramètre `ColorIndex` ou `Font` du Label, nous ne la détaillerons donc pas non plus ici.

Le *PicturePushButton* est un widget interactif, il implémente donc 4 méthodes pour traiter les événements provenant de la souris (`processMouseDown`, `processMouseReleased`, `processMouseClicked` et `processMouseDown`) et 3 méthodes pour traiter les événements provenant du clavier (`processValidationKey`, `processAbortKey` et `processNormalKey`). Le comportement du *PicturePushButton* n'est pas impacté par les événements du clavier ni par les mouvements du curseur : seul les événements `mousePressed` et `mouseClicked` influence son comportement. Ce comportement est représenté en Figure 5.32. Ainsi, on peut constater que, si le *PicturePushButton* est visible (jeton dans la place `Visible`) et actif (jeton dans la place `Enabled`), il va traiter les clics qu'il reçoit. Ainsi, il traite tout d'abord l'événement `mouseDown` (méthode `processMouseDown`) en changeant l'état du *PicturePushButton* de relâché (jeton dans la place `ButtonReleased`) à pressé (jeton dans la place `ButtonPressed`). Lorsque l'utilisateur relâche le bouton de la souris, le serveur va appeler la méthode `processMouseClicked` sur le *PicturePushButton*. Dans ce cas, le *PicturePushButton* va envoyer l'événement `A661_EVT_SELECTION` et remettre le *PicturePushButton* dans l'état relâché.

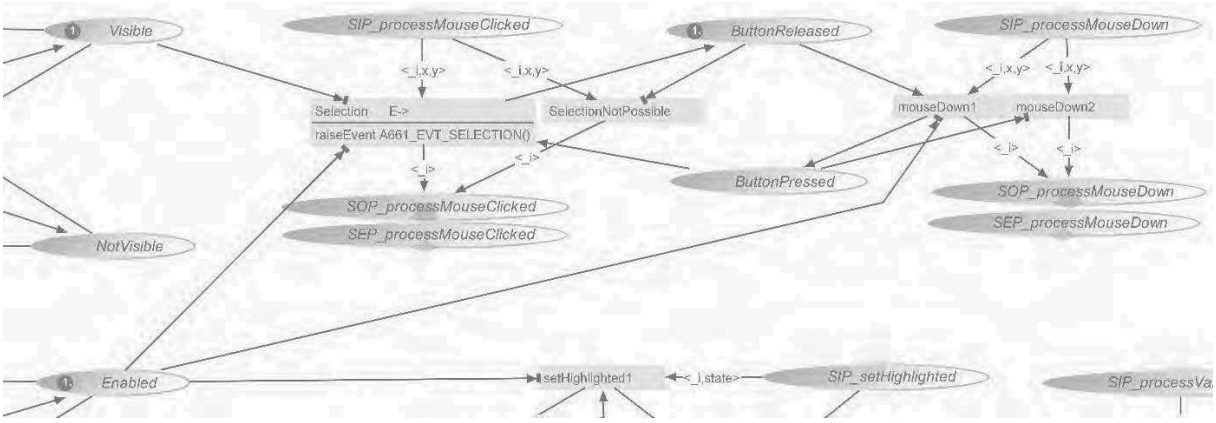


Figure 5.32. Modèle ICO de la gestion du clic pour un PicturePushButton

Le fait que le *PicturePushButton* soit un widget interactif implique également qu'il a un paramètre modifiable en cours d'exécution supplémentaire : le paramètre `Highlighted`. Celui-ci permet de mettre en évidence le widget lorsqu'il est survolé par le curseur. Le comportement de la modification de ce paramètre est présenté en Figure 5.33.

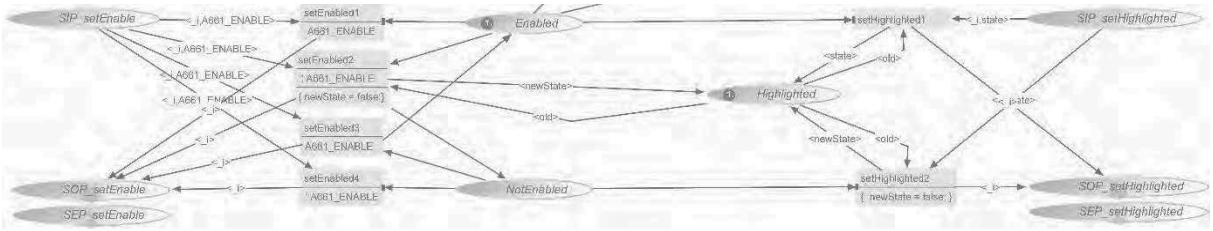


Figure 5.33. Modèle ICO de la gestion du paramètre Highlighted

Lors d'un appel de la méthode `setHighlighted`, deux cas de figure sont possibles :

- Le widget est actif (jeton dans la place `Enabled`) : dans ce cas la valeur du paramètre est modifiée dans la place `Highlighted` au franchissement de la transition `setHighlighted1`.
- Le widget est inactif (jeton dans la place `NotEnabled`) : dans ce cas, le widget ne peut pas avoir le focus et la valeur du paramètre est placée à `false`.

L'introduction de ce paramètre implique la modification de la gestion du paramètre `Enable`. En effet, il est important de garantir que lorsque le widget est désactivé (transition `setEnabled2`), la valeur de son paramètre `Hilighted` passe à `false`.

5.4.5 Modélisation du serveur

Le serveur correspond aux composants logiciels de l'interaction physique : il est composé du serveur graphique et du gestionnaire de fenêtres (voir section 4.3.4). Concrètement, il est responsable de la gestion des événements provenant du périphérique d'entrée (le KCCU) et du rendu graphique sur le périphérique de sortie (l'écran).

Le serveur est décomposé en plusieurs composants logiciels correspondants à différentes fonctionnalités (voir section 4.3.2.1) : les drivers, le calcul des coordonnées absolues, le calcul des techniques d'interactions, de la sélection d'objets graphiques (*picking*) et le rendu graphique. Le serveur de notre application, pour garantir la compatibilité avec le standard ARINC 661, contient en plus un composant appelé graphe de scène ou `SceneGraph` qui est responsable de la hiérarchie des widgets. C'est ce composant qui est utilisé pour la sélection d'objets graphiques ainsi que pour le rendu graphique global de l'application.

Le serveur étant un modèle très important, nous ne pouvons le présenter en entier dans ce document pour des raisons de lisibilité. Cependant, les sous-sections suivantes détaillent les différents composants cités ci-dessus ainsi que la partie du modèle qui leur est associée. Il est à noter que nous ne présentons pas ici le modèle des drivers car celui-ci est hors du périmètre de notre étude (voir section 4.1.2). Nous ne présentons pas non plus le composant en charge du calcul des coordonnées absolues. En effet, celui-ci est très dépendant des drivers et n'a que très peu d'intérêt s'il est présenté sans ses composants. Cependant, les travaux de (Ladry 2010) montrent que ces deux composants peuvent être modélisés à l'aide de la notation ICO.

5.4.5.1 Interaction Techniques : calcul des techniques d'interactions

Dans le cas de notre application, une seule technique d'interaction est disponible, elle correspond au clic de la souris. Ainsi, le composant en charge du calcul des techniques d'interactions est en charge du calcul du clic de la souris. Celui-ci est effectué par le modèle ICO présenté en Figure 5.34. Dans ce cas, le serveur envoie un événement clic lors de la réception d'un événement `mousePressed` suivi de celle d'un événement `mouseReleased`.

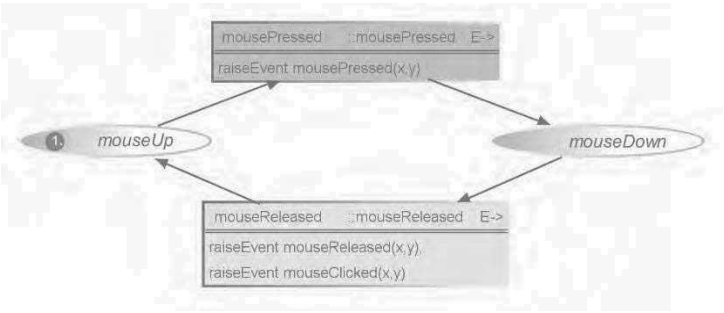


Figure 5.34. Modèle ICO de la gestion du clic

Nous présentons ici une technique d'interaction très simple car, pour des raisons de sûreté de fonctionnement, les techniques d'interactions sont actuellement très limitées par le standard ARINC 661. Cependant la notation ICO rend possible la modélisation de techniques d'interactions beaucoup plus compliquées comme nous le montrent les travaux de (Ladry 2010) et de (Hamon 2014).

5.4.5.2 Picking : sélection d'objets graphiques

Le *Picking* est responsable de la sélection d'objet graphique, c'est-à-dire de l'identification des widgets ciblés par les actions de l'utilisateur. Son rôle peut être divisé en deux :

- *Gestion du focus des widgets* : lors du mouvement de la souris, il détecte si un widget interactif est présent sous le manipulateur de souris (le curseur graphique). Si c'est le cas, il fait appel à la fonction `setHighlighted(true)` du widget pour demander la modification de son état (voir section 5.4.4.4). Lorsque le manipulateur de souris n'est plus situé sur un widget, le *Picking* fera alors appel à la fonction `setHightlighted(false)` du widget.
- *Gestion des événements provenant des actions de l'utilisateur autres que le mouvement de la souris* : lors de la réception d'un événement correspondant à une action utilisateur autre que le mouvement de la souris, le *Picking* transfère cet événement au widget situé sous le manipulateur de souris si celui-ci est interactif et a obtenu le focus.

Il est important de noter qu'un serveur respectant le standard ARINC 661 et les spécifications d'Airbus propose à certains widgets une fonctionnalité de *Caging*. Nous ne présentons pas cette fonctionnalité ici car elle n'est pas utilisée par les widgets que nous avons présentés[1]. Cependant cette fonctionnalité influe sur le comportement du Picking qui n'est effectué que lorsque le mode *Caging* du serveur est désactivé. La désactivation de ce mode est représentée par la présence d'un jeton dans les places virtuelles `CagingClosed` dans les modèles que nous avons présenté dans le présentons ci-dessous (Figure 5.35 et Figure 5.36).

Gestion du focus des widgets

La modélisation en ICO du comportement dédié à la gestion du focus des widgets est représentée en Figure 5.35.

Si le mode *Caging* du serveur est désactivé (jeton dans la place `CagingClosed`), la réception d'un évément `mouseMove(x,y)` provoquera le franchissement de la transition `receiveMouseMove`. Ce franchissement place un jeton contenant les coordonnées correspondantes à cet événement dans la place `coordinateForMouseMove`. Ce jeton contient également un index `i` provenant de la place `IndexIn`. Il est incrémenté pour chaque réception d'événement `mouseMove(x,y)`. Cet index permet de garantir que les événements sont traités dans leur ordre d'arrivée : le franchissement de la transition `pickWidgetForMouseMove` se fait en fonction de l'index placé dans la place `IndexOut` qui est incrémenté lors de la fin du traitement de l'événement.

Le franchissement de la transition `pickWidgetForMouseMove` réalise une appel de la méthode `findWidgetAt(x,y)` du *SceneGraph*. Cet appel de méthode retourne un identifiant qui est égal à l'identifiant du widget si un widget interactif, actif et visible est situé à la position `(x,y)`. Si ce n'est pas le cas, le *SceneGraph* retourne comme identifiant la valeur `-1`. Le franchissement de cette transition retire le jeton situé dans la place `IndexOut`. Celui-ci est replacé lors de la fin du traitement de cet événement, ce qui rendra possible le traitement de l'événement suivant.

Après incrément de l'index (franchissement de la transition `incrementIndex`), un jeton, contenant l'index de sortie `j` et l'identifiant du widget `id`, est placé dans la place `WidgetToPick`. La place `WidgetToPick` contient donc l'identifiant du widget qui doit obtenir le focus, alors que la place `PickedWidget` contient l'identifiant du widget qui a le focus. Ces deux identifiants peuvent être égaux à `-1` et correspondent dans ce cas à l'absence de widget ayant ou devant obtenir le focus. Quatre cas de figure sont alors envisageables pour l'actualisation du widget ayant le focus (représenté par l'identifiant du jeton dans la place `PickedWidget`) :

- Le widget qui doit obtenir le focus (jeton dans la place `WidgetToPick`) correspond à celui qui a déjà le focus (jeton dans la place `PickedWidget`). Dans ce cas, aucune action supplémentaire n'est effectuée et le franchissement de la transition `discardPickingChange` replace le nouvel index dans la place `IndexOut`.

[1] Le lecteur désireux d'en savoir plus à propos de cette fonctionnalité pourra se reporter au Chapitre 9 dans lequel elle est détaillée.

- Aucun widget n'a le focus (`id=-1` dans la place `PickedWidget`) et un widget doit obtenir le focus. Dans ce cas, la transition `changePickedWidget1` est franchie en effectuant un appel de méthode `setHighlighted(true)` sur le widget qui doit obtenir le focus.
- Un widget a le focus et l'identifiant correspondant au widget qui doit obtenir le focus est égal à -1, ce qui signifie que le focus à venir ne concerne aucun widget. Dans ce cas, la transition `changePickedWidget2` est franchie et effectuant un appel de méthode `setHighlighted(false)` sur le widget qui avait le focus jusqu'à présent.
- Un widget a le focus et un nouveau widget doit obtenir le focus. Dans ce cas, la modification se fait en deux étapes : premièrement, le franchissement de la transition `changePickedWidget3` appelle la méthode `setHighlighted(false)` sur le widget qui a le focus et deuxièmement le franchissement de la transition `changePickedWidget4` appelle la méthode `setHighlighted(true)` sur le widget qui doit obtenir le focus.

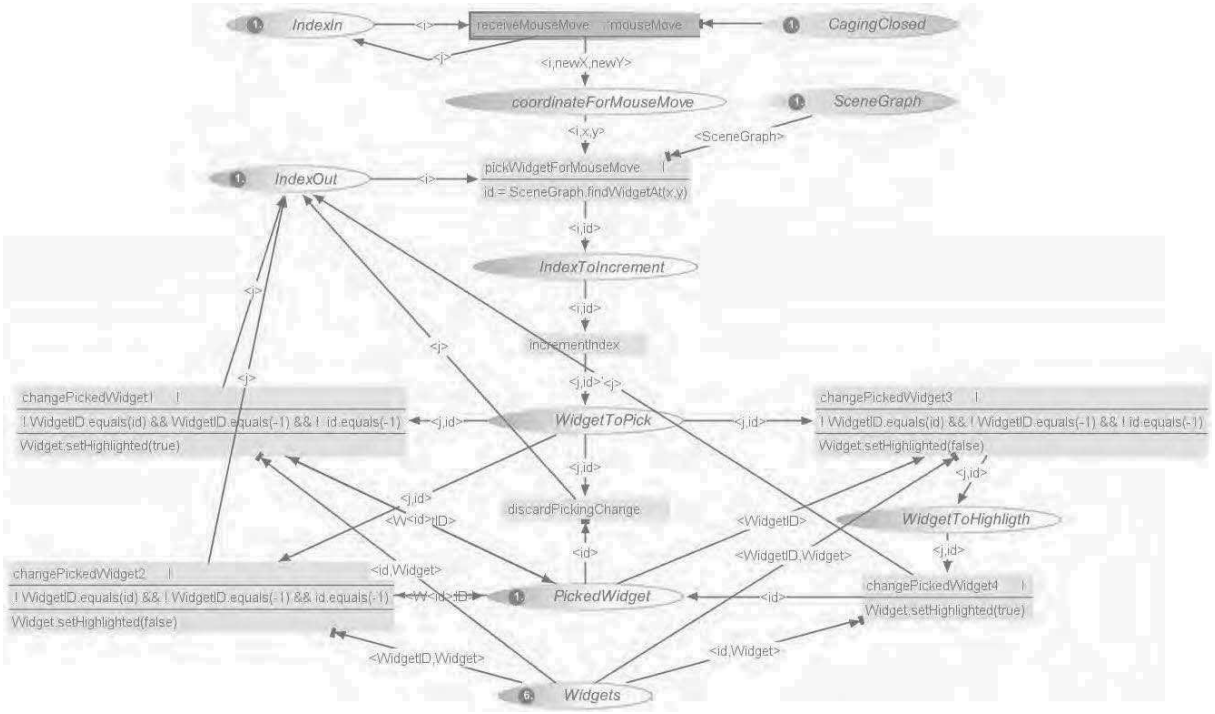


Figure 5.35. Extrait du modèle ICO du serveur : gestion du focus des widgets sur réception des événements `mouseMove` provenant des actions de l'utilisateur

Gestion des événements provenant des actions de l'utilisateur autres que le mouvement de la souris

La modélisation en ICO du comportement dédié à la gestion des événements provenant des actions de l'utilisateur autres que le mouvement de la souris est représentée en Figure 5.36.

Ce comportement peut-être divisé en deux : la gestion des événements provenant de la souris (`mousePressed`, `mouseReleased` et `mouseClicked`) et la gestion des événements provenant du clavier (`handleValidationKey`, `handleNormalKey` et `handleAbortKey`).

Pour la gestion des événements provenant de la souris, nous prenons l'exemple de l'événement `mouseClicked`. Lors de la réception de cet événement, si le mode *Caging* du serveur est désactivé (jeton dans la place `CagingClosed`), la transition `receiveMouseClicked` est franchie et place un jeton contenant les coordonnées correspondantes à l'événement `(x,y)` dans la place `coordinateForMouseClicked`. Suit alors le franchissement de la transition `pickWidgetForMouseClicked` qui appelle la méthode `findWidgetAt(x,y)` sur le *SceneGraph*. Cette méthode retourne l'identifiant `id` du widget situé aux coordonnées `(x,y)` si un widget interactif, actif et visible est situé à cet endroit (autrement `id = -1`). Le franchissement de cette transition (`pickWidgetForMouseClicked`) place un jeton dans la place

widgetToNotifyMouseClicked. Si l'identifiant contenu dans ce jeton est différent de -1 et correspond à celui du widget qui a le focus (identifiant idPicked contenu par le jeton présent dans la place PickedWidget), le serveur transmet le click au widget concerné par un appel de la méthode processMouseClicked. Ceci est fait grâce au franchissement de la transition processMouseClickedOnWidget. Si aucun widget n'est concerné on ne tient pas compte de cet événement (franchissement de la transition discardMouseClickedNotOnInteractiveWidget).

Ce comportement est similaire pour la gestion des événements mousePressed et mouseReleased. Pour des raisons de lisibilité et car cet événement n'est pas utilisé par les widgets que nous avons présentée précédemment, nous n'avons pas représenté la gestion de l'événement mouseReleased sur le modèle présenté en Figure 5.36.

La gestion des événements provenant du clavier est effectuée par les transitions handleValidationKey, handleNormalKey et handleAbortKey. Prenons l'exemple d'une pression sur la touche de validation. Cette action de l'utilisateur déclenche l'envoi de l'événement handleValidationKey. Lors de la réception de cet événement, la transition handleValidationKey est franchie si le mode Caging du serveur est désactivé (jeton dans la place CagingClosed) et si un widget a le focus (correspondance entre l'identifiant contenu dans le jeton de la place PickedWidget et l'identifiant contenu dans l'un des jetons de la place Widgets qui rassemble tous les widgets de l'application). Le franchissement de cette transition implique l'appel de la méthode processValidationKey() sur le widget ayant le focus.

La gestion des événements handleNormalKey et handleAbortKey est similaire à celle que nous venons de présenter.

5.4.5.3 SceneGraph : gestion de la hiérarchie des widgets

Le *graphe de scène* (ou *SceneGraph*) est responsable de la gestion de la hiérarchie de widgets. Il regroupe toutes les caractéristiques de l'ensemble des widgets nécessaires à la fois au rendu graphique et au picking.

Plus particulièrement, il est responsable du calcul de la visibilité et l'activation finale des widgets ainsi que du calcul de leur position absolue sur l'écran (ces trois paramètres dépendants de la valeur des paramètres de leurs parents, voir section 3.1.2.2). Il regroupe également tous les éléments spécifiques pour le rendu graphique de chaque widget et est abonné à leur changement. Par exemple, pour un *PicturePushButton*, il est abonné aux modifications de la valeur des paramètres Highlighted, PictureReference, LabelString et StyleSet.

Le modèle du SceneGraph étant particulièrement imposant et n'illustrant aucune nouvelle technique de modélisation, nous le décrivons dans le Chapitre 9 qui présente la modélisation complète d'une application de taille réelle à l'aide de la notation ICO.

5.4.5.4 Renderer : gestion du rendu graphique

Le composant de rendu (Renderer) gère l'affichage général de l'application ainsi que celui du curseur graphique correspondant à la trackball du KCCU. Il est donc abonné aux modifications d'états du composant SceneGraph.

Le composant de rendu permet de faire le rendu graphique de l'application. Il n'est pas modélisé à l'aide de la notation formelle ICO. Dans notre cas, ce composant correspond à une classe utilisant la libraire Swing pour faire le dessin final de l'application.

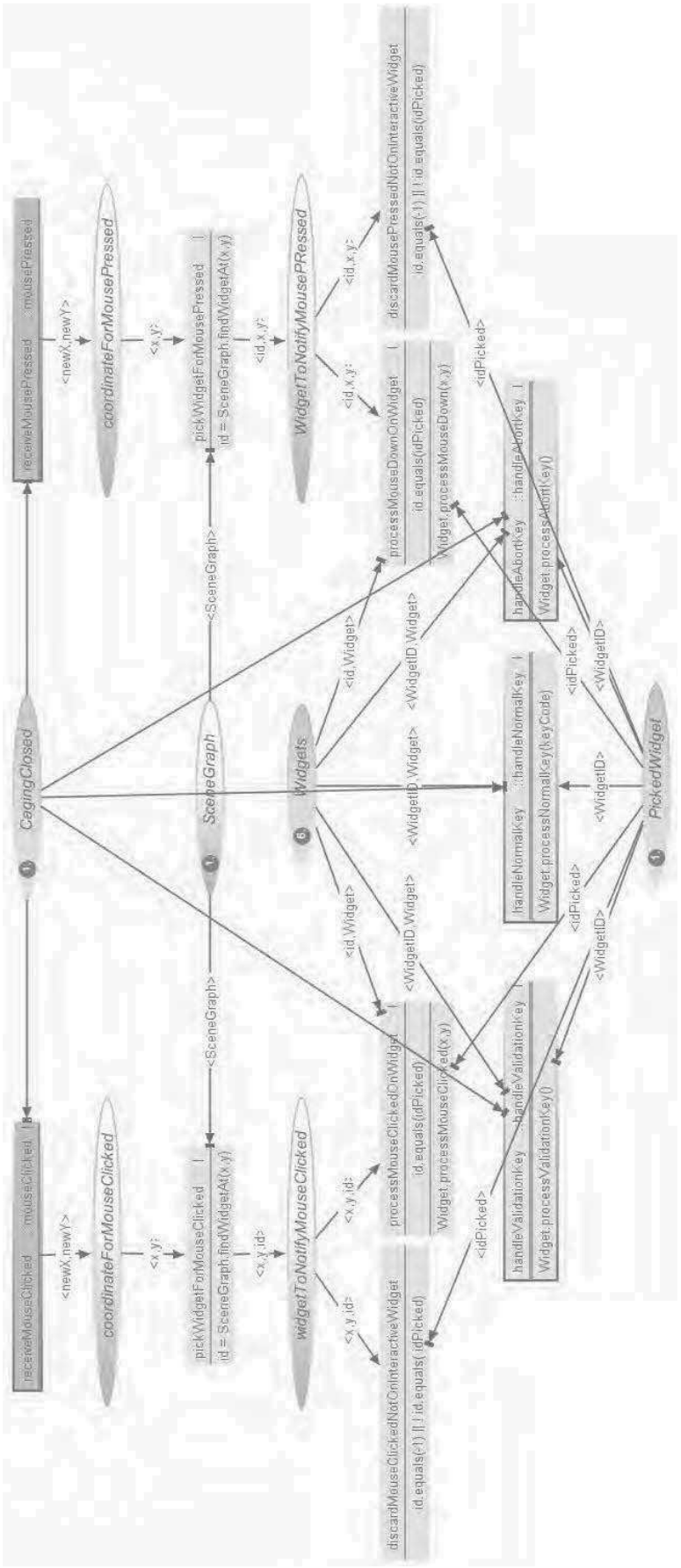


Figure 5.36. Extrait du modèle ICO du serveur : gestion des événements provenant des actions utilisateurs (autres que le `mouseMove`)

5.5 Conclusion

Ce chapitre a proposé une approche vers une conception zéro-défaut des systèmes interactifs à travers un développement par les modèles à l'aide d'une technique de description formelle des différents composants logiciels d'un système interactif.

Pour cela, nous avons présenté la notation formelle ICO que nous utilisons dans ce but ainsi que les différents éléments qui la composent et comment ceux-ci nous permettent de modéliser les aspects essentiels des systèmes interactifs.

Nous avons également présenté comment notre approche à base de modèles peut-être appliquée à l'architecture générique pour les systèmes interactifs que nous avons présentée en section 4.3.

Afin de mettre en évidence les techniques utilisées et les concepts permettant de modéliser tous les composants logiciels d'un système interactif, nous avons illustré l'application de notre approche à base de modèles sur un exemple simple : l'application « les 4 saisons ».

Chapitre 6. Architecture logicielle générique pour des systèmes interactifs tolérants aux fautes

Sommaire

Malgré tous les efforts portés lors du développement du système, nous avons montré que des fautes naturelles pouvaient affecter les systèmes interactifs, imposant alors l'utilisation de mécanismes de tolérance aux fautes pour les traiter. Ce chapitre présente donc une architecture logicielle générique et tolérante aux fautes pour les systèmes interactifs. Cette architecture est fondée sur l'application du principe de la programmation *n*-autotestable à tous les composants logiciels des systèmes interactifs.

Nous présentons en premier lieu une discussion sur les différentes architectures logicielles tolérantes aux fautes et leur applicabilité aux composants interactifs ainsi que les raisons qui nous ont poussé au choix de fonder notre architecture sur la programmation *n*-autotestable. Ainsi, le lecteur désireux de connaître notre architecture et ne s'intéressant pas à ces considérations est invité à se rendre directement à la troisième section.

La première section présente en premier lieu trois architectures logicielles classiques pour la tolérance aux fautes et pouvant être appliquées aux différents composants logiciels des systèmes interactifs : les architectures autotestables, n-autotestables et NVP. Pour chacune de ces architectures, nous étudions son applicabilité aux composants logiciels des systèmes interactifs ainsi que les avantages et inconvénients.

La deuxième section présente les raisons de notre choix pour un couple d'architectures particulièrement adaptées aux systèmes interactifs dans les cockpits avioniques : les architectures autotestables et n-autotestables.

Enfin, la troisième section présente notre architecture logicielle générique et tolérante aux fautes pour les systèmes interactifs.

Dans un premier temps comment appliquer une architecture autotestable avec contrôleur d'assertions aux systèmes interactifs présents dans les cockpits avioniques. Afin de permettre la réalisation de cette architecture, nous proposons une méthode pour la définition des contrôleurs d'assertions s'appuyant sur l'utilisation d'une analyse des modes de défaillance des composants logiciels concernés. Cette analyse des modes de défaillance suit la méthode des AMDEC et s'effectue à l'aide d'une description de l'architecture logicielle du système interactif avec le langage graphique AADL et de diagrammes de séquence représentant le comportement du système.

Dans un second temps, nous présentons une architecture logicielle autotestable appliquée aux composants logiciels d'un système interactif avionique. Ces composants sont capables de notifier les

pilotes en cas de détection d'une erreur. Dans ce cas, ceux-ci peuvent décider d'utiliser un composant redondant au CDS pour la commande et le contrôle des systèmes critiques de l'avion.

Dans un troisième temps, nous présentons une architecture logicielle n-autotestable permettant de réaliser le recouvrement automatique de certaines erreurs dans le CDS sans impliquer les pilotes.

6.1 Applicabilité des architectures de tolérance aux fautes aux composants interactifs

En nous appuyant sur les mécanismes de tolérance aux fautes les plus répandus (présentés en section 2.6), nous proposons dans cette section trois architectures logicielles de tolérance aux fautes pouvant être appliquées aux systèmes interactifs. Nous étudions plus particulièrement les trois mécanismes suivants : le mécanisme d'autotestabilité, le mécanisme de *n*-autotestabilité et la programmation en n-version. Nous n'étudions pas le mécanisme des blocs de recouvrement car celui-ci crée, en cas d'erreur, des suspensions de service pouvant se révéler longues, ce qui serait inacceptable du point de vue de l'utilisateur (voir section 2.6.2).

Les trois architectures présentées s'appuient sur le principe de redondance du composant que l'on souhaite rendre tolérant aux fautes (aussi appelé composant fonctionnel). Suivant le but escompté, cette redondance peut-être réalisée de différentes manières :

- En utilisant de la *réplication* : les composants redondants sont des copies du composant fonctionnel) ;
- En utilisant de la *diversification* : les composants redondants sont des variantes diversifiées du composant fonctionnel) ou encore, pour les architectures autotestables et n-autotestables,
- En utilisant un *contrôleur d'assertion* : cette option est applicable seulement dans le cas des architectures autotestables et *n*-autotestable. Dans ce cas, le composant redondant est un contrôleur d'assertions permettant de vérifier le comportement du composant fonctionnel.

Nous présentons dans un premier temps, en section 6.1.1 les trois architectures de tolérance aux fautes et leur principe de fonctionnement. Dans un second temps, nous étudions en section 6.1.2 l'applicabilité de ces architectures aux systèmes interactifs et à leurs différents composants logiciels.

6.1.1 Architectures tolérantes aux fautes

6.1.1.1 Architecture autotestable

Un composant autotestable (Laprie, Deswartes, et al. 1996), est capable de vérifier son propre fonctionnement et de notifier une erreur si celui-ci n'est pas correct. Pour cela, un composant est chargé de vérifier le comportement et l'exécution du composant fonctionnel (le composant que l'on souhaite rendre tolérant aux fautes). Le mécanisme de tolérance aux fautes autotestable est également appelé COM-MON, notamment dans l'industrie avionique (Traverse, Lacaze et Souyris 2004). Cette dénomination provient de la redondance du composant fonctionnel (le composant `COM` pour commande) par le composant qui vérifie son exécution (le composant `MON` pour moniteur).

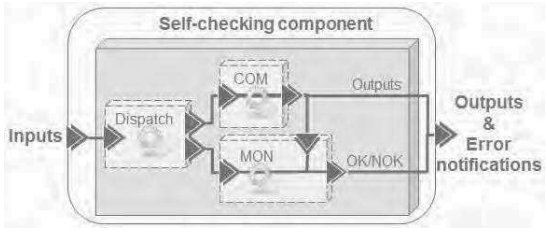


Figure 6.1. Architecture logicielle d'un composant autotestable

La Figure 6.1 présente l'architecture logicielle correspondant à un composant autotestable. Elle nous permet de mettre en évidence les trois composants qui le constituent :

- *Le composant* `COM` (COMmande). C'est le composant fonctionnel, celui que l'on souhaite rendre tolérant aux fautes.
- *Le composant* `MON` (MONitor). C'est le composant en charge de la vérification du comportement du composant `COM`.
- *Le composant* `Dispatch` : c'est le composant en charge de délivrer les entrées du composant autotestable aux composants `COM` et `MON`.

L'instanciation de cette architecture soulève deux problèmes de réalisation : celle du composant `Dispatch` et celle du composant `MON`. Nous identifions trois manières différentes de réaliser ce dernier :

- *Redondance* : le composant `MON` est composé d'une **copie** du composant `COM` (appelée `Contrôleur`) et d'un composant `Comparateur` en charge de vérifier que les résultats du `COM` et de sa copie sont similaires.
- *Diversification* : le composant `MON` est composé d'une **variante** du `COM` (appelée `Contrôleur`) et d'un composant `Comparateur` en charge de vérifier que les résultats du `COM` et de sa variante sont similaires.
- *Contrôleur d'assertion* : le composant `MON` est un contrôleur d'assertions. Une assertion est une expression logique décrivant une partie du comportement attendu du composant COM. Un contrôleur d'assertion est un test booléen permettant de vérifier la concordance entre les assertions et les résultats du composant COM. Le résultat du test prend la valeur vraie si la vraisemblance est validée autrement, il prend la valeur fausse et un signal d'erreur est déclenché.

Il est important de rappeler qu'un composant autotestable ne permet que la détection des erreurs et ne permet pas leur recouvrement. Celui-ci peut-être alors effectué de deux manières : premièrement, un composant spécifique peut être développé pour traiter les erreurs détectées ; deuxièmement, on peut utiliser de la redondance de composants autotestables, ce qui correspond à l'architecture *n*-autotestable présentée dans la section suivante.

6.1.1.2 Architecture n-autotestable

Un composant *n*-autotestable (Laprie, Arlat, et al. 1990) est fondé sur la redondance de composants autotestables. Cette redondance permet le recouvrement des erreurs qui ont été détectées par les composants autotestables.

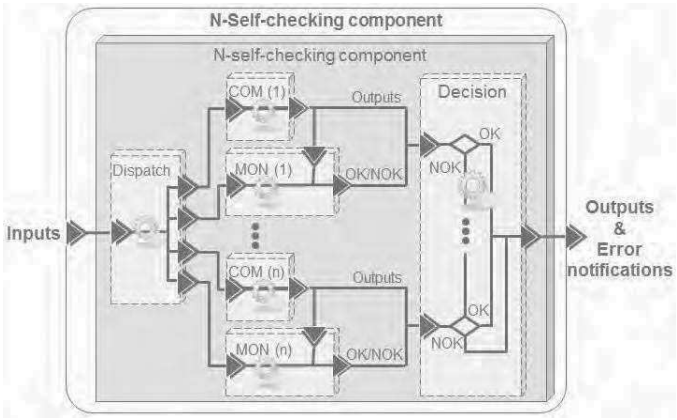


Figure 6.2. Architecture logicielle d'un composant n-autotestable

La Figure 6.2 présente l'architecture logicielle d'un composant n-autotestable. Elle est composée de *n* composants autotestables identifiés par un numéro de séquence. Les entrées du composant n-autotestable sont distribuées par le composant `Dispatch` aux composants `COM` et `MON` de chaque composant autotestable. Tous les composants autotestables traitent en parallèle ces entrées et fournissent un résultat accompagné de sa notification d'erreur (qui valide ou invalide le résultat) au composant `Decision`. Ce composant observe le résultat de chaque composant autotestable, en suivant leur numéro de séquence. Ainsi, si le composant autotestable n°1 fournit un résultat valide, le composant `Decision` envoie ce résultat comme étant celui du composant n-autotestable. Si, au contraire, le résultat du composant

autotestable n°1 n'est pas valide, le composant `Decision` s'intéresse au résultat du composant autotestable suivant et ainsi de suite. S'il s'avère que les résultats de tous les composants autotestables ont été invalidés, le composant `Decision` envoie alors le résultat du dernier composant autotestable avec une notification d'erreur.

L'instanciation de cette architecture soulève trois problèmes de réalisation : celle du composant `Dispatch`, celle de la redondance des composants autotestables et celle du composant `Decision`. La redondance des composants autotestables peut être réalisée de différentes manières. Premièrement, nous pouvons utiliser des copies d'un composant autotestable (quelle que soit l'option de réalisation du composant autotestable choisie). Deuxièmement, nous pouvons, utiliser la diversification de tous les composants `COM` ou `MON` ; ou la diversification des composants `COM` et `MON`. Troisièmement, nous pouvons, dans le cas d'un composant autotestable utilisant un contrôleur d'assertions, diversifier le composant `COM` et utiliser un contrôleur d'assertion unique.

6.1.1.3 Architecture NVP

Un composant NVP (N-Version-Programming) s'appuie sur les principes de la programmation en n-versions (Avizienis 1985) qui est fondée sur la redondance et la diversification du composant fonctionnel.

La Figure 6.3 présente l'architecture logicielle d'un composant NVP. Cette architecture est composée de n versions du composant fonctionnel. Ces versions sont des variantes diversifiées du composant fonctionnel. Nous proposons cependant de pouvoir les remplacer par des copies de celui-ci. Dans ce cas, on se rapproche des principes de la redondance active (Chereque, et al. 1992) ou du mécanisme de Triple-Modular-Redundancy ou TMR (Lyons et Vanderkulk 1962) qui correspond à une architecture 3-VP utilisant des copies du composant fonctionnel. Les entrées du composant NVP sont distribuées aux différentes variantes par le composant `Dispatch`. Les versions traitent ces entrées en parallèle et envoient leur résultat au composant `Vote`. Celui-ci est en charge de déterminer le résultat majoritaire qui est considéré comme correct et est envoyé comme résultat du composant NVP. Il est important de noter que pour que le composant `Vote` puisse déterminer un résultat majoritaire en cas de dysfonctionnement d'une des variantes, il est nécessaire d'avoir un nombre de variantes supérieur à deux et impair.

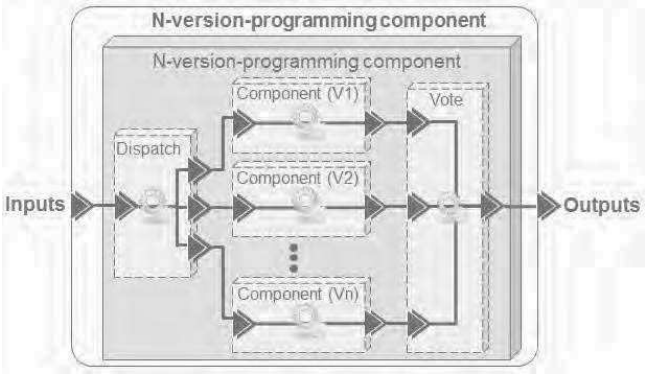


Figure 6.3. Architecture logicielle d'un composant NVP

6.1.2 Applicabilité des architectures aux systèmes interactifs

Les trois architectures que nous avons présentées permettent la tolérance des fautes matérielles ou la tolérance des fautes logicielles et matérielles. En effet, ces architectures sont fondées sur la redondance du composant que l'on souhaite rendre tolérant aux fautes. Si l'on se contente de cette redondance, nous ne pouvons tolérer que les fautes matérielles. Par contre, si les différents composants redondants sont diversifiés nous pouvons également tolérer les fautes logicielles.

Pour que ces architectures logicielles soient efficaces, il est important de ségréguer les différents composants. La ségrégation de composants consiste à les isoler spatialement et temporellement à la manière du partitionnement spatial et temporel défini par le standard ARINC 653 (AEEC 2003). Celui-

ci permet d'assurer qu'il n'y a pas de propagation d'erreur entre les différents composants, autrement dit, qu'une faute affectant un composant ne se propagera pas en erreur dans un autre composant, ce qui pourrait perturber la détection et le recouvrement des erreurs. Cette problématique de propagation d'erreur concerne particulièrement les fautes naturelles ; cependant, elle concerne également certaines fautes logicielles telles que des buffer overflow ou des pointeurs invalides.

Nous nous intéressons à présent à l'applicabilité de ces architectures pour les systèmes interactifs : dans quelle mesure celles-ci peuvent être instanciées pour les différents composants logiciels des systèmes interactifs ? Nous nous intéressons plus particulièrement aux trois composants logiciels que nous avons identifiés pour les systèmes interactifs dans les cockpits avioniques et présentés en Figure 4.15 : le serveur, les widgets et l'UA (pour User Application).

Nous avons pointé du doigt dans la section précédente que les difficultés de réalisation des architectures présentées étaient dans la réalisation du ou des composants redondants ; celle du composant de distribution des entrées et celle des différents composants de comparaison, décision ou vote. Lorsque nous nous intéressons à de la réplication, les composants redondants sont des copies du composant fonctionnel, cette solution ne présente donc aucun problème au niveau de la réalisation des composants redondants à la différence de la solution de diversification et de celle du contrôleur d'assertion. Nous présentons donc dans un premier temps comment effectuer ces deux solutions de redondance pour les trois composants identifiés (le serveur, les widgets et l'UA) ; dans un second temps, nous présentons également des remarques quant à la réalisation des autres composants (comparaison, décision ou vote).

6.1.2.1 Réalisation de serveurs redondants

Diversification

Le serveur est un composant compliqué avec beaucoup de fonctionnalités. La diversification d'un tel composant est donc très coûteuse. En effet, elle nécessite la création d'une spécification complète et non ambigüe du serveur ainsi que le développement de variantes par des équipes de développement différentes, en utilisant des algorithmes, des langages et des outils de programmation différents.

Si l'on considère des systèmes déjà existants, nous pouvons envisager de réaliser la diversification d'un serveur de deux manières. Une première solution pourrait être de considérer les serveurs de deux versions d'un même système d'exploitation (par exemple deux versions d'une distribution Ubuntu. Une seconde solution pourrait être de considérer les serveurs de deux systèmes d'exploitation différents. Il est cependant nécessaire de s'assurer que les spécifications haut-niveau restent similaires sinon la comparaison ou le vote de leurs sorties n'aurait aucun sens et ce travail peut s'avérer très fastidieux.

Contrôleur d'assertions

Cette option de mise en œuvre est particulièrement adaptée au serveur du fait de la complexité et des nombreuses fonctionnalités de ce composant. En effet, à partir des spécifications du serveur, il est possible de décider celles qui sont essentielles pour sa sûreté de fonctionnement et celles qui sont secondaires. À partir des spécifications des fonctionnalités qui auront été identifiées comme essentielles, il est nécessaire de déduire des assertions qui doivent être toujours vraies en cas de bon fonctionnement de ces fonctionnalités. Nous pouvons ensuite réaliser des contrôleurs qui vérifient que les résultats du serveur ne violent pas les assertions.

Un contrôleur d'assertion est généralement plus petit qu'une variante d'un composant car il ne s'occupe que de la vérification des fonctionnalités identifiées comme essentielles. C'est un avantage non négligeable dans les systèmes contraints où l'espace mémoire et la puissance du processeur sont limités, par exemple, dans les systèmes embarqués dans les avions.

6.1.2.2 Réalisation de widgets redondants

Diversification

Dans le cas des widgets WIMP, il peut être facile d'obtenir des variantes en utilisant des versions déjà existantes de widgets provenant de différentes bibliothèques de widgets (*toolkits*). Dans ce cas, la difficulté est transférée (comme pour le serveur) dans la réalisation du composant en charge de la comparaison ou du vote sur les sorties. Cependant, la réalisation d'un tel composant sera simplifiée par rapport à celle du serveur car les différentes bibliothèques de widgets respectent le standard CUA (Common User Access) d'IBM (IBM 1989).

Dans le cas où l'on souhaite rendre tolérants aux fautes des widgets non répandus (par exemple, des widgets définis par le standard ARINC 661 (AEEC 2013)), il est alors nécessaire de créer une variante de ce widget. Pour cela, une spécification complète et non-ambigüe du comportement du widget est nécessaire. Celle-ci permet le développement d'une variante du widget par une autre équipe, en utilisant des langages et des outils différents.

Contrôleur d'assertions

De même que dans le cas du serveur, cette option impose d'étudier la spécification de chaque type de widget pour en déduire les fonctionnalités essentielles pour la sûreté de fonctionnement. Pour chacune d'elles, nous définissons une assertion qui reste vraie si son comportement est correct. Ainsi, la vérification de ces assertions permet de contrôler le comportement du widget. Elles permettent notamment de contrôler que le widget traite correctement les événements provenant des utilisateurs et les changements d'états associés.

6.1.2.3 Réalisation d'UA redondantes

Diversification

Tout comme le serveur, les UA sont des composants compliqués et spécifiques. Ainsi, la création d'une variante requiert une spécification complète et non ambigüe et sera très coûteuse.

Contrôleur d'assertions

Du fait de la taille et la complexité des UA, cette option est particulièrement adaptée. Dans ce cas, les assertions sont majoritairement fondées sur le comportement du contrôleur de dialogue, autrement dit le comportement sémantique du système interactif. Les assertions sont donc très dépendantes du système interactif et de son domaine d'application.

6.1.2.4 Réalisation des composants additionnels

Les architectures de tolérance aux fautes que nous avons présentées s'appuient, en plus des composants redondants (qu'ils soient de simples copies, des variantes ou des contrôleurs d'assertion), sur des composants supplémentaires qui posent également des problèmes de réalisation :

- *Le composant* `Dispatch` : il est responsable de la distribution des entrées du composant tolérant aux fautes et les composants redondants. La complexité dans la réalisation de ce composant réside dans le fait de distribuer les entrées de manière synchrone et fiable afin de s'assurer que les composants redondants traitent les mêmes entrées et que, par conséquent, leurs sorties sont comparables. Ce composant peut-être supprimé si l'on utilise un protocole de diffusion fiable distribuant les entrées aux différents composants redondants (on peut prendre l'exemple du réseau AFDX (AEEC 2009) associé à un protocole atomique FIFO).
- *Le composant* `Comparateur` : il est spécifique à l'architecture autotestable et à l'architecture *n*-autotestable utilisant deux variantes ou copies du même composant. Il est responsable de la comparaison entre les sorties des composants redondants. Dans le cas de deux copies ou de deux variantes issues des mêmes spécifications ce composant est assez trivial : en effet, il doit faire la comparaison entre deux sorties et de renvoyer une erreur en cas d'incohérence. Dans le cas de deux

variantes existantes, la réalisation de ce composant peut-être un peu plus compliquée car il peut y avoir besoin d'analyser les sorties pour pouvoir les comparer.

- *Le composant* `Decision` : il est spécifique à l'architecture *n*-autotestable. Son comportement, et donc par conséquent sa réalisation, est assez simple. En effet, il est responsable de prendre les sorties des différents composants autotestables dans l'ordre qui leur a été affecté et d'envoyer le résultat (si celui-ci est valide) ou de s'intéresser à celui du composant autotestable suivant. Si tous les composants autotestables sont défaillants, ils fournissent tous une sortie invalidée. Dans ce cas, le composant `Decision` envoie une notification d'erreur.
- *Le composant* `Vote` : il est spécifique à l'architecture NVP. Les problématiques soulevées par sa réalisation sont similaires à celles du composant Comparateur. Ainsi, si les composants redondants sont des copies ou des variantes issues des mêmes spécifications, il est juste responsable de faire un vote majoritaire sur les différentes sorties. Cependant, dans le cas de deux variantes existantes, la réalisation de ce composant peut-être un peu plus compliquée car cela peut nécessiter d'analyser les sorties pour pouvoir les comparer.

6.1.3 Remarques conclusives sur les différentes architectures proposées

Pour chaque architecture tolérante présentée, nous avons étudié trois options de réalisation différentes :

- *Réplication* : les différents composants redondants sont des copies du composant original.
- *Diversification* : les différents composants redondants sont des variantes du composant original.
- *Contrôleur d'assertions* : cette option concerne seulement l'architecture autotestable et l'architecture n-autotestable. Dans leur cas, on peut remplacer un des deux composants redondants par un composant spécifique vérifiant le comportement du composant original à travers la vérification d'assertions représentant leur comportement.

Architectures \ Spécificités		Tolérance aux fautes		Fautes couvertes		Variantes/Copies pour tolérer f fautes	Autres composants nécessaires	Partitions nécessaires (ségrégation)
		Détection	*Recouvrement*	*Matérielles*	*Logicielles*			
Autotestable	*Réplication*	✓	×	✓	×	2 copies (f = 1)	1 dispatch 1 comparateur	3
	Diversification	✓	×	✓	✓	2 variantes (f = 1)	1 dispatch 1 comparateur	3
	Contrôleur d'assertions	✓	×	✓	✓	1 composant (f = 1)	1 dispatch 1 contrôleur d'assertions	3
N-autotestable	*Réplication*	✓	✓	✓	×	2(f+1) copies	1 dispatch 1 decision	2(f+1)+2
	Diversification	✓	✓	✓	✓	2(f+1) variantes	1 dispatch f+1 comparateurs 1 decision	2(f+1)+2
	Contrôleur d'assertions	✓	✓	✓	✓	f+1 variantes	1 dispatch 1 contrôleur d'assertions 1 decision	2(f+1)+2
NVP	*Réplication*	✓	✓	✓	×	f+2 copies	1 dispatch 1 vote	f+4
	Diversification	✓	✓	✓	✓	f+2 variantes	1dispatch 1 vote	f+4
	Contrôleur d'assertions	n/a	n/a	n/a	n/a	n/a	n/a	n/a

✓ : *Couvert* | × : *Non couvert*

Tableau 6.1. Récapitulatif des spécificités des différentes architectures

Le Tableau 6.1 présente les architectures étudiées et leurs spécificités. Nous pouvons ainsi voir que les architectures utilisant la réplication permettent seulement de traiter les fautes matérielles alors que les autres architectures permettent de traiter à la fois les fautes matérielles et les fautes logicielles.

Bien que les différentes architectures autotestables permettent seulement la détection des erreurs (et pas leur recouvrement), elles sont un bon point de départ pour rendre les systèmes interactifs tolérants aux fautes car les composants autotestables sont la base des architectures n-autotestables. De plus, dans le cas des systèmes interactifs, le recouvrement des erreurs peut également être confié à l'utilisateur.

Les architectures n-autotestable et NVP demandent beaucoup plus de ressources en terme de partitions (voir colonne `Partitions nécessaires`). Cependant, elles permettent de tolérer plus de fautes.

Le Tableau 6.2 présente le coût d'applicabilité de ces architectures aux différents composants de notre architecture. Ce tableau distingue les composants répandus pour lesquels il existe déjà de nombreuses variantes tels que les widgets répondant aux spécifications du standard CUA (colonnes `COTS` pour Commercial On-The-Shelf) et les composants peu répandus pour lesquels il faudra obligatoirement développer ses propres variantes tels que les widgets répondant aux spécifications du standard ARINC 661 (colonnes `Spécifique`). Les architectures utilisant de la réplication sont les moins coûteuses, car elles ne demandent aucun type de diversification logicielle (qui est très couteuse).

La diversification est plus coûteuse pour les composants compliqués tels que le serveur et l'application interactive, elle est d'autant plus coûteuse pour les architectures *n*-autotestable et NVP car ces architectures nécessitent un plus grand nombre de variantes que l'architecture autotestable : le développement d'une variante coute entre 0,75 et 0,85 fois le coût de développement du composant original (Laprie, Arlat, et al. 1990).

Enfin, l'utilisation d'un contrôleur d'assertion dans le cas des architectures autotestables et *n*-autotestables permet de réduire les coûts de développement par rapport à la diversification du composant.

Architectures \ Composants		Serveur		Widgets		Application Interactive	
		COTS	*Spécifique*	*COTS*	*Spécifique*	*COTS*	*Spécifique*
Autotestable	*Réplication*	★	★	★	★	n/a	★
	Diversification	★★★	★★★★	★	★★	n/a	★★★★
	Contrôleur d'assertions	★★	★★	★★	★★	n/a	★★
N-autotestable	*Réplication*	★	★	★	★	n/a	★
	Diversification	★★★★	★★★★★	★	★★★	n/a	★★★★★
	Contrôleur d'assertions	★★★	★★★	★★★	★★★	n/a	★★★
NVP	*Réplication*	★	★	★	★	n/a	★
	Diversification	★★★★	★★★★★	★	★★★	n/a	★★★★★
	Contrôleur d'assertions	n/a	n/a	n/a	n/a	n/a	n/a

Difficulté d'applicabilité croissante : ★ | ★★ | ★★★ | ★★★★ | ★★★★★ (- → +)

Tableau 6.2. Récapitulatif de l'applicabilité de chaque architecture aux différents composants interactifs en termes de difficulté et de coût

6.2 Choix d'une architecture de tolérance aux fautes adaptée au contexte de la thèse

Nous avons étudié les différentes architectures de tolérance aux fautes et la manière dont elles pourraient être appliquées aux composants des systèmes interactifs. Cependant, dans le cadre de cette thèse, nous nous intéressons plus particulièrement aux systèmes interactifs dans les cockpits avioniques. Comme nous l'avons vu dans le Chapitre 3, le développement de ces systèmes interactifs doit respecter de nombreuses contraintes. Les systèmes interactifs embarqués dans le cockpit doivent notamment être compatibles avec le standard ARINC 661 (AEEC 2013). De plus, ils doivent également être développés en respectant un niveau d'assurance de développement élevé (DAL B ou DAL A). Ainsi, très peu de compagnies peuvent développer de tels systèmes : ils ne peuvent donc pas être considérés comme des COTS (voir section 6.1.3) et la diversification logicielle sera d'autant plus coûteuse du fait de ces contraintes de développement.

Notre modèle de faute (voir section 4.2.3) comprend à la fois les fautes logicielles de développement et les fautes matérielles impactant le système en opération. Les fautes matérielles en opération sont inévitables et imprédictibles (Taber et Normand 1993). Elles ne peuvent donc pas être traitées comme les fautes logicielles par prévention et élimination comme nous l'avons proposé dans le Chapitre 5. Le seul moyen de traiter ces fautes et d'éviter les défaillances qui leur sont consécutives est

de tolérer les erreurs dont elles sont la cause. De plus, bien que la méthode proposée au Chapitre 5 permette de limiter autant que possible les fautes logicielles, il est pratiquement impossible de créer un système parfait sans aucune faute de développement (Hamilton 1986). Pour pouvoir garantir le plus haut niveau de sûreté de fonctionnement de nos systèmes, nous choisissons donc ici une architecture capable de tolérer à la fois les fautes logicielles de développement résiduelles et les fautes matérielles en opération.

Enfin, les utilisateurs des systèmes interactifs que nous étudions dans cette thèse (concrètement, les pilotes de l'avion) sont des utilisateurs formés à l'utilisation des systèmes et aux procédures à effectuer en cas de défaillance de l'un d'eux.

Toutes ces spécificités nous ont poussés à choisir de nous appuyer sur l'architecture autotestable avec contrôleur d'assertions. En effet, celle-ci permet la détection des erreurs dues à la fois aux fautes matérielles en opération et aux fautes logicielles de développement. Concernant le recouvrement des erreurs, celui-ci peut être soit effectué par les utilisateurs, soit effectué en utilisant les composants autotestables comme la base d'une architecture *n*-autotestable. De plus, elle est beaucoup moins coûteuse que les autres architectures permettant de détecter et tolérer les erreurs dues à ces deux types de fautes (architecture autotestable avec diversification ou architecture NVP). Les architectures autotestables et *n*-autotestables ont également prouvé leur utilité et applicabilité pour les systèmes embarqués critiques car elles sont utilisées pour de nombreux systèmes avioniques embarqués tels que les calculateurs de commande de vol (Traverse, Lacaze et Souyris 2004).

6.3 Architecture logicielle pour des systèmes interactifs tolérants aux fautes

Pour rendre les systèmes interactifs tolérants aux fautes, nous nous appuyons, pour la détection des erreurs, sur une architecture autotestable avec contrôleur d'assertions et pour le recouvrement des erreurs, nous proposons deux options : confier le recouvrement à l'utilisateur (au pilote) ou s'appuyer sur une architecture *n*-autotestable.

La complexité ici réside dans l'instanciation de l'architecture autotestable aux composants interactifs. Plus particulièrement, elle réside dans la réalisation des différents contrôleurs d'assertions, autrement dit des composants responsables de la détection des erreurs : les composants MON. Pour cela, nous proposons d'abord une méthode pour la réalisation des composants interactifs autotestables. Puis nous proposons d'intégrer les différents composants autotestables dans une architecture logicielle. Enfin, pour le recouvrement des erreurs, nous proposons de rendre les systèmes interactifs tolérants aux fautes grâce à l'utilisation de mécanismes spécifiques à la gestion des erreurs ou grâce à l'instanciation d'une architecture n-autotestable.

6.3.1 Composants interactifs autotestables

La complexité de l'application de l'architecture autotestable avec contrôleur d'assertions aux différents composants de nos systèmes interactifs, repose dans la définition du contrôleur d'assertions. Il s'agit donc de définir les assertions représentant le comportement attendu de chaque composant et de définir ensuite le moyen de surveiller ses assertions, de vérifier qu'elles sont respectées. Pour cela, nous proposons trois étapes :

- **Étape 1 : Définition et analyse du système**

  Identification de l'architecture, des composants et des fonctions du système interactif. Pour cela, nous nous appuyons sur la description de l'architecture physique et logicielle du système à l'aide du langage architectural AADL et la description des fonctions des différents composants à l'aide de diagrammes de séquences.

- **Étape 2 : Identification des modes de défaillance**

  Identification systématique des modes de défaillance de toutes les fonctions des composants du système. Pour cela, nous nous appuyons sur une Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité (une AMDEC ou FMECA en anglais (Department of the Army 2006))…

- **Étape 3 : Identification des assertions et définition des mécanismes de surveillance**

  Identification des assertions associées aux défaillances identifiées en étape 2. Ces assertions sont ensuite formalisées et servent de fondement pour la définition de mécanismes permettant de les surveiller.

Afin d'expliciter notre méthode, nous détaillons dans les sous-sections suivantes les trois étapes présentées ci-dessus en nous appuyant sur un exemple concret.

6.3.1.1 Définition et analyse du système

Dans un premier temps, il est nécessaire de définir le système qui nous intéresse, ses composants et les relations entre les différents composants afin de pouvoir identifier les modes de défaillance que l'on cherche à éviter ainsi que les assertions qui leur sont associées.

Pour cela, nous proposons dans un premier temps d'identifier les composants du système qui nous intéressent ainsi que leurs fonctionnalités et les liaisons qui les relient entre eux. Nous proposons de réaliser cette identification à travers la description du système avec le langage architectural graphique AADL (SAE International 2012), que nous complétons par un ou plusieurs diagrammes de séquence représentant les connexions entre les différents composants lors d'un scénario d'exécution.

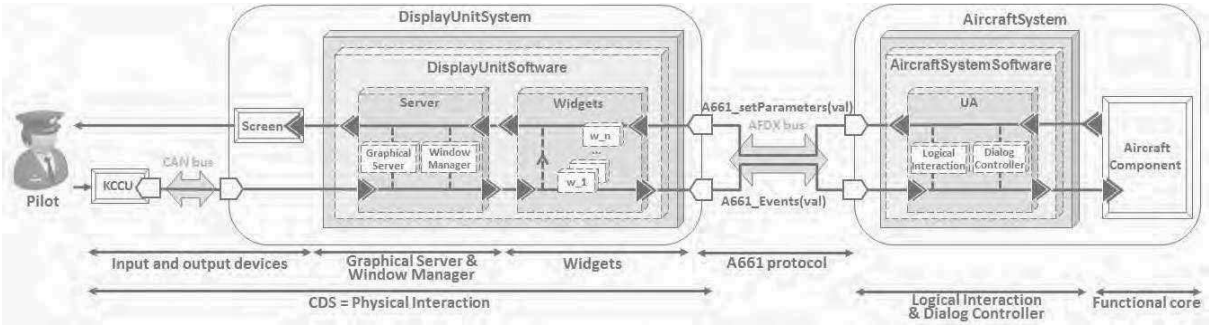


Figure 6.4. Architecture simplifiée d'un cockpit interactif respectant le standard ARINC 661

Nous proposons ici de décrire l'architecture générique haut-niveau des systèmes interactifs dans les cockpits d'avions civil. La Figure 6.4 présente la représentation de l'architecture générique haut niveau d'un système interactif de cockpits avionique en AADL (voir section 4.3.4).

Cette représentation haut niveau de notre système nous permet de mettre en évidence les composants principaux sur lesquels nous fondons notre diagramme de séquence : le serveur et les widgets qui forment le CDS) et l'UA. Nous rappelons ci-dessous le rôle des différents composants de cette architecture (de droite à gauche) :

- *Les périphériques d'entrée et de sortie :* le KCCU et l'écran, qui permettent l'interaction entre le système et l'équipage.
- *Le serveur :* il est composé du serveur graphique et du gestionnaire de fenêtres. Il est responsable de la gestion des évènements provenant du KCCU et du rendu graphique de l'application sur l'écran. Il gère notamment l'affichage du curseur graphique et de la distribution des événements correspondants aux actions de l'utilisateur aux widgets ciblés. Il est également responsable du calcul des techniques d'interaction et de la gestion du graphe de scène (la hiérarchie des widgets).
- *Les widgets :* ce sont les composants interactifs de base. Ils correspondent aux éléments d'affichage et d'interaction pour une UA (User Application). Ainsi, ils permettent aux pilotes de déclencher des actions sur les systèmes avioniques et ils permettent à l'UA de notifier aux pilotes les actions qu'ils peuvent effectuer.

- *L'User Application (UA) :* c'est l'interface logicielle des systèmes avioniques. Elle permet de traiter les actions de l'équipage sur les widgets à travers la réception des événements qu'ils envoient (appelés `A661_Events(val)`). L'UA est également responsable de notifier l'équipage d'un changement d'état du système avionique à travers des appels de méthodes pour modifier l'aspect graphique et l'état des widgets (appelées `A661_setParameters(val)`).
- *Les systèmes avioniques :* ce sont les systèmes physiques de l'avion (par exemple, les moteurs…).

Le serveur et les widgets ont un fonctionnement générique, ils réagissent de la même manière quel que soit le système avionique concerné. Par exemple, le serveur traite toujours de la même manière un événement clic correspondant à une action du pilote en s'intéressant à l'emplacement de ce clic sur l'écran et en le transmettant au widget concerné si un widget actif et visible se situe sous ce clic. De la même manière, lors de la réception d'un clic, un PicturePushButton réagit toujours de la même manière en envoyant (s'il est bien actif et visible) un événement `A661_EVT_SELECTION` à l'UA. Contrairement au serveur et aux widgets, le comportement de l'UA (qui régit le comportement de l'interface) est très dépendant du système avionique.

Du fait de cette organisation, nous illustrons plus particulièrement la création des contrôleurs d'assertions sur les parties génériques du système interactif (le serveur et les widgets). En effet, cela nous permet de rester générique et ne nécessite pas d'instancier notre méthode sur une application particulière. De plus, le fait de s'intéresser à ces deux composants particuliers permet de créer un CDS (Control and Display System) tolérant aux fautes qui pourra ensuite être réutilisé quel que soit le système avionique concerné et donc quelle que soit l'UA. Il est cependant important de souligner que tous les principes que nous présentons peuvent s'appliquer de la même manière à l'UA.

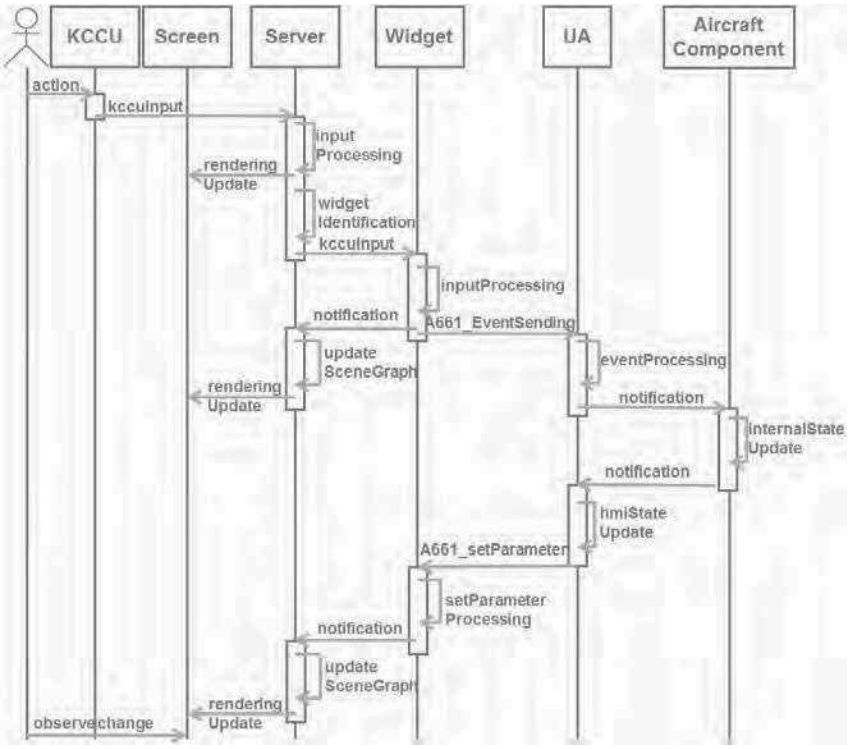


Figure 6.5. Diagramme de séquence haut-niveau du fonctionnement du système interactif dans le cockpit

Pour pouvoir identifier les modes de défaillance des différents composants du système interactif et de leurs fonctions, nous proposons de représenter son comportement à l'aide d'un diagramme de séquence UML. La Figure 6.5 représente une abstraction haut-niveau du comportement du système interactif à travers un diagramme de séquence. Pour simplifier la lecture, nous n'avons représenté sur cette figure qu'un seul widget. Ce diagramme de séquence décrit le comportement du système interactif suivant :

- Lorsqu'un pilote effectue une action sur un widget à travers l'utilisation du KCCU (`action`), le serveur identifie cette action (`inputProcessing`), met à jour la position du curseur graphique

(renderingUpdate), identifie le widget ciblé (widgetIdentification) et transmet l'action de l'utilisateur à ce widget. Le widget traite cette action en fonction de son état (inputProcessing) et notifie l'UA de l'action de l'utilisateur au travers de l'envoi d'un événement A661_Event(val). Le widget notifie également, s'il y a lieu, le serveur de son changement d'état (notification) ; ce qui permet au serveur de mettre à jour le graphe de scène (updateSceneGraph) et l'affichage (renderingUpdate). Enfin, l'UA traite l'événement envoyé par le widget (eventProcessing) et notifie le système avionique qui modifie son état en fonction du contrôle désiré par l'utilisateur (internalStateUpdate).

Exemple : *l'utilisateur clique sur un bouton interactif pour engager un cap à 30° (cap qu'il a déjà rentré au préablable). Le serveur identifie le widget ciblé et lui transmet le clic. Le widget traite ce clic et envoie un événement adapté à l'UA qui le traite et demande au système avionique d'engager le cap à 30°.*

- À chaque modification de l'état interne d'un système avionique (internalStateUpdate), celui-ci informe l'UA de son changement d'état (notification). L'UA met alors à jour son état interne (hmiStateUpdate) et demande au widget concerné d'actualiser son état en conséquence au travers de l'appel d'une méthode setParameter (A661_setParameter(val)). Le widget met alors à jour son état tel que demandé par l'UA (setParameterProcessing) et notifie le serveur de ce changement d'état (notification). Enfin, le serveur met à jour le graphe de scène (updateSceneGraph) et l'affichage (renderingUpdate).

  Exemple : *le système avion passe dans un état où le changement de cap n'est plus accessible pour le pilote. L'UA analyse ce changement d'état et décide de désactiver le bouton qui permet au pilote de demander l'engagement d'un nouveau cap. L'UA fait un appel de méthode sur le widget pour lui demander de passer en état désactivé (setEnable(false)). Le widget met ensuite à jour son état et notifie le serveur qui modifie le graphe de scène et l'affichage du widget (en le grisant pour notifier au pilote sa désactivation).*

6.3.1.2 Identification des modes de défaillance

Dans un second temps, nous analysons les modes de défaillance du système interactif en nous appuyant sur les différents diagrammes de séquence tels que celui présenté en Figure 6.5. Pour identifier les défaillances qui peuvent affecter notre système, nous effectuons une AMDEC (Analyse des Modes de Défaillance, leurs Effets et leur Criticité) (Department of the Army 2006).

1	2	3	4	5	6	7	8
Item	Failures modes	Potential causes	Local effects	Upper-level effects	Risk level	Safety mechanisms (SM)	Upper-level effect with SM

Tableau 6.3. Trame d'un tableau d'AMDEC

L'AMDEC est un procédé d'analyse des modes de défaillances des systèmes. Son objectif est de déterminer de manière systématique, pour toutes les fonctions d'un composant tous les modes de défaillance qui peuvent l'affecter. Les résultats d'une AMDEC sont traduits dans une table, respectant la trame présentée dans le Tableau 6.3, et récapitulant pour chaque fonction ses modes de défaillance, leurs effets et leur criticité. Ainsi, une AMDEC est représentée comme un tableau de huit colonnes. La colonne 1 représente les objets auxquels on s'intéresse. Ces objets peuvent être des objets bas-niveau tels que les fonctions d'un composant ou des objets plus haut-niveau tel qu'un composant. Pour chaque objet, nous identifions les différents modes de défaillance qui peuvent l'affecter (colonne 2). Chaque mode de défaillance est associé aux causes qui ont pu le créer (colonne 3), ses effets au niveau du composant (colonne 4), ses effets au niveau du système (colonne 5) et sa criticité (colonne 6). La criticité correspond au degré de dangerosité du mode de défaillance et de ses effets sur le système et ses utilisateurs à la manière des degrés de criticité définis par la norme CS-25 (voir section 3.2.1). Les colonnes 7 et 8 présentent les mécanismes de sûreté de fonctionnement mis en place pour limiter les effets du mode de défaillance ainsi que l'effet final après la mise en place des mécanismes de sûreté de fonctionnement (qui est supposé nul si les mécanismes sont suffisamment efficaces).

1	2	3	4	5	6
Item	**Failures modes**		**Local effects**	**Upper-level effects**	**Consequence classification**
Server.inputProcessing Identify the kccu input and update cursor rendering	No execution Server.inputProcessing.FM1		Upon the receipt of a kccu input event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution Server.inputProcessing.FM2		Upon the receipt of a kccu input event, the server forwards the wrong kccu input event	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution Server.inputProcessing.FM3		The server processes a kccu input event to without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
Server.widgetIdentification Identify the targeted widget, and forward of the inputEvent	No execution Server.widgetIdentification.FM1		Upon the receipt of a kccu input event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution Server.widgetIdentification.FM2		Upon the receipt of a kccu input event, the server forwards it to a wrong widget	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution Server.widgetIdentification.FM3		The server sends a kccu input event to a widget without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
Server.updateSceneGraph Update the scene graph and the graphical rendering of the application	No execution Server.updateSceneGraph.FM1		Upon the receipt of an update notification, the server does not update the scene graph or the graphical rendering	The pilot is not notified of the aicraft system state change	Loss of data display
	Erroneous execution Server.updateSceneGraph.FM2		Upon the receipt of an update notification, the server updates the scene graph and/or the graphical rendering in a wrong way	The pilot is notified of a wrong aicraft system state change	Erroneous data display
	Unexpected execution Server.updateSceneGraph.FM3		The server updates the scene graph or the graphical rendering without receiving any update notification	The pilot is notified of an aircraft system state change that did not occur	Erroneous data display
Widget.inputProcessing Process the click, send the corresponding A661_Event(val) to the UA and send an update notification to the server	No execution Widget.inputProcessing.FM1		Upon the receipt of a kccu input event, the widget does not send any A661_Event(val)	The pilot command is not sent to the aicraft system	Loss of control
	Erroneous execution Widget.inputProcessing.FM2		Upon the receipt of a kccu input event, the widget sends a wrong A661_Event(val) (or with a wrong value)	A wrong command is sent to the aircraft system	Erroneous control
	Unexpected execution Widget.inputProcessing.FM3		The widget sends an A661_Event(val) without receiving any kccu input event	A command is sent t the aircraft system without any user action	Erroneous control
Widget.setParameterProcessing Process A661_setParameter(val) and send an update notification to the server	No execution Widget.setParameterProcessing.FM1		The widget does not update its corresponding parameter	The pilot is not notified of the aicraft system state change	Loss of data display
	Erroneous execution Widget.setParameterProcessing.FM2		The widget updates its corresponding parameter with a wrong value	The pilot is notified of a wrong aicraft system state change	Erroneous data display
	Unexpected execution Widget.setParameterProcessing.FM3		The widget updates a parameter without receiving an A661_setParameter	The pilot is notified of an aircraft system state change that did not occur	Erroneous data display

Tableau 6.4. AMDEC résumant les défaillances génériques du serveur et des widgets

1	2	3	4	5	6
Item	Failures modes		Local effects	Upper-level effects	Failure classification

Tableau 6.5. Trame de l'AMDEC générique pour les systèmes interactifs

Pour faire une AMDEC générique pour les systèmes interactifs, nous avons légèrement modifié la trame classique, celle que nous utilisons est représentée au Tableau 6.5.

Nous ne présentons pas la colonne 3 car nous considérons dans notre cas que les modes de défaillances identifiés peuvent être causés par les fautes que nous avons identifiées dans notre modèle de faute (voir section 4.2.3). Cette colonne serait ainsi uniquement constituée de cases comprenant la même cause : les fautes logicielles de développement et/ou les fautes matérielles en opération affectant le système durant son fonctionnement.

Nous avons également retiré les colonnes 7 et 8 car nous ne connaissons pas, à cette étape, les mécanismes de sûreté de fonctionnement qui seront appliqués. En effet, il est important de rappeler que le but de cette AMDEC est de définir les modes de défaillance des différents composants afin de pouvoir déterminer les assertions qui nous serviront à construire nos composants autotestables.

Enfin, nous avons également modifié la colonne 6 de ce tableau. Nous présentons dans cette colonne la classification du mode de défaillance par rapport aux quatre classes de modes de défaillances que nous avons identifiées en section 4.2.2. En effet, nous ne pouvons pas à ce niveau d'abstraction présenter la criticité du mode de défaillance d'une fonction car cette criticité correspond à une combinaison de la classe de défaillance (que nous présentons en colonne 6) et de la sémantique du système avionique concerné.

Ainsi, par exemple, la perte de tous les affichages de l'altitude de l'avion serait classifiée de défaillance catastrophique selon l'échelle définie par la norme CS-25 alors que la perte de l'affichage du cap stabilisé serait classifiée de défaillance majeure. Nous considérons donc dans cette colonne quatre classes de défaillances :

- *Perte de contrôle* : le contrôle demandé par l'utilisateur n'est pas effectué.
- *Contrôle erroné* : le contrôle effectué ne correspond pas à celui demandé par l'utilisateur.
- *Perte d'affichage* : l'affichage correspondant au changement d'état du système n'est pas effectué.
- *Affichage erroné* : l'affichage correspondant à l'état du système est effectué de manière inadéquate.

Le Tableau 6.4 présente l'AMDEC générique que nous avons effectuée sur les fonctions du serveur et des widgets en nous appuyant sur les fonctions identifiées dans le diagramme de séquence présenté en Figure 6.5.

6.3.1.3 Identification et formalisation des assertions et de leur contrôleur

Pour définir les assertions décrivant le comportement attendu du système, nous nous appuyons sur l'AMDEC du système. Ainsi, à chaque mode de défaillance identifié correspond une ou plusieurs assertions. Ces assertions sont tout d'abord exprimées de manière textuelle et doivent être formalisées afin de pouvoir être contrôlées. Nous proposons de les formaliser sous forme d'expressions logiques. À chaque assertion, nous faisons correspondre un contrôleur d'assertion. Pour définir un tel contrôleur d'assertion, il est nécessaire de déterminer un certain nombre de paramètres :

- *Les observables* : ils décrivent les données qui doivent transiter entre le composant fonctionnel (dont on vérifie l'exécution) et le contrôleur d'assertion. Ils correspondent par exemple au résultat de l'exécution d'une fonction, l'état interne ou la valeur des données internes du composant fonctionnel.
- *L'image de l'état* : elle représente les données qui doivent être stockées dans le composant contrôleur afin de pouvoir exécuter le contrôle d'assertion (par exemple, l'ancienne position du curseur est nécessaire pour la vérification du calcul de sa nouvelle position). Elle correspond à un état simplifié du composant fonctionnel et est mise à jour régulièrement en fonction des entrées reçues par le contrôleur.
- *La périodicité* : elle permet de déterminer si le contrôleur d'assertion est exécuté de manière périodique ou apériodique. Dans le premier cas, il est exécuté toutes les *n* périodes d'exécution du

logiciel. Dans le second cas, son exécution est conditionnée par la réception d'un événement déclencheur.

- *Le déclencheur* : il définit l'élément qui déclenche l'exécution du contrôleur d'assertion. Dans le cas d'un contrôle périodique, le déclencheur est le début de la *n*-ième période. Dans le cas d'un contrôle apériodique, le déclencheur sera un événement tel que la réception d'un événement correspondant à une action de l'utilisateur ou à l'exécution d'une fonction.

Pour permettre la classification des différentes assertions et de leur contrôleur, nous ajoutons une identification pour chaque assertion et chaque contrôleur ainsi que pour le mode de défaillance auquel est associée l'assertion. Nous pouvons également noter qu'une assertion peut avoir une portée plus importante que celle d'un seul mode de défaillance ; elle peut ainsi être associée à plusieurs modes de défaillance de la même fonction.

Prenons l'exemple du second mode de défaillance de la fonction `server.widgetIdentification` du serveur. Cette fonction permet de détecter le widget ciblé par l'action du pilote, autrement dit, le widget, actif et visible, qui se situe à la même position que le curseur. Le second mode de défaillance de cette fonction (appelé `server.widgetIdentification.FM2`) correspond à l'exécution erronée de cette fonction. Il correspond donc à l'identification d'un autre widget que celui réellement ciblé par l'action. L'assertion correspondant à ce mode de défaillance (nommée A1) ainsi que son contrôleur (nommé AM1) sont décrits dans le Tableau 6.6.

Assertion Definition	*Id*	A1
	Item	server.widgetIdentification
	Textual definition	A1 is the assertion that proves that the widget identified by the server is the right one. To be the correct one, the widget identified (w) must fulfil three conditions: i) it must be in the right state to receive user input, which means that it must be visible ($w.visibility = true$) and enabled ($w.enabling = true$); ii) it must be positioned under the user cursor ($e.(x,y) \subset w.zone$); iii) the event processed by the server must correspond to the user action ($e=u.action()$)
	Formal definition	Let s be a Server, let u be a User, let W be the set of widgets of the application and let E be the set of KCCU events $\forall w \in W, \forall e \in E, w = s.widgetIdentification(e)$ $\Leftrightarrow$ $w.visibility = true \wedge w.enabling = true \wedge e.(x,y) \subset w.zone \wedge e = u.action( )$
Assertion Monitor	*Id*	AM1
	Failure mode	server.widgetIdentification.FM2
	Textual definition	w fulfils all the conditions required for A1 (i.e. it must be visible, enable and under the cursor position).
	Observables	widgetIdentified
	State image	i) the old cursor position and ii) $P(w)$, the set of parents of w and their state.
	Frequency	Aperiodic
	Trigger	Reception of an event e corresponding to a user action

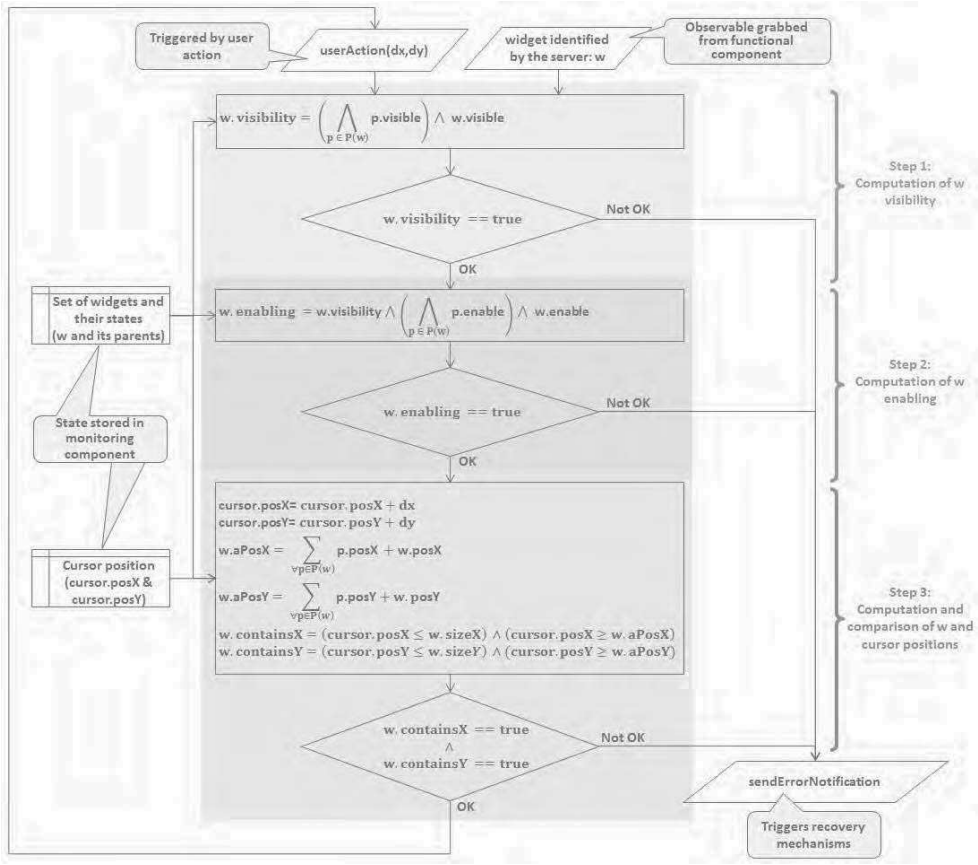
Tableau 6.6. Tableau définissant l'assertion A1 et son contrôleur AM1

A1 est donc une assertion garantissant que le widget identifié par la fonction `widgetIdentification` du serveur est bien le widget ciblé par l'action de l'utilisateur. Pour cela, il est nécessaire que le widget identifié par la fonction soit à la fois visible ($w.visibility = true$), actif ($w.enabling = true$), qu'il se situe à la même position que le curseur au moment de la réception de l'événement ($e.(x,y) \subset w.zone$) et que l'événement traité par le serveur corresponde bien à celui fait par l'utilisateur ($e = u.action( )$).Cette assertion se traduit par la définition formelle exprimée dans la ligne « Assertion/Formal definition » du Tableau 6.6.

Le contrôleur de l'assertion A1, nommé AM1, doit vérifier que le widget w, identifié par le serveur du composant fonctionnel (le composant `COM`), vérifie bien toutes les propriétés nécessaires pour être identifié par le serveur. Celui-ci doit être à la fois visible, actif et situé sous le curseur. Pour pouvoir vérifier cela, le contrôleur d'assertion doit connaître l'identifiant *widgetIdentified* du widget identifié par le serveur. Il doit également stocker l'ancienne position du curseur ainsi que son propre arbre des

widgets, image de l'arbre des widgets du composant COM. C'est sur cet arbre qu'il effectue les calculs pour vérifier que l'image (*w*) du widget identifié par le serveur (*widgetIdentified*) remplit bien toutes les conditions pour satisfaire A1. Cette vérification est déclenchée à chaque réception d'un événement utilisateur. Elle est également présentée sous forme algorithmique avec l'organigramme associé en Figure 6.6. Il est ainsi possible de distinguer les trois étapes de cette vérification :

- Étape 1 : calcul de la visibilité de *w'*, si celui-ci n'est pas visible, le contrôleur envoie une erreur.
- Étape 2 : calcul de l'activation de *w'*, si celui-ci n'est pas actif, le contrôleur envoie une erreur.
- Étape 3 : calcul de position de *w'* et de celle du curseur, si les deux ne sont pas cohérentes, le contrôleur envoie une erreur.



Algorithmic description	if (event == userAction(dx,dy) & widgetIdentified== w){
	if (w.visibility != true) {errorDetected = 1;}
	if (w.enabling != true) {errorDetected = 1;}
	if (userAction.(x,y) ⊂ w.zone != true) {errorDetected = 1;}
	send(widgetIdentified, errorDetected);
	}

Figure 6.6. Organigramme correspondant à l'exécution du contrôleur d'assertion AM1

6.3.2 Architecture logicielle pour la détection des erreurs

Nous proposons à présent d'intégrer ces contrôleurs dans une architecture logicielle qui permet la détection des erreurs dans le système interactif, et plus particulièrement dans le CDS (le serveur et les widgets).

Cette architecture, présentée en Figure 6.7, met en évidence les composants COM et MON du CDS. Afin de limiter le nombre de partitions nécessaires pour la ségrégation des différents composants, nous avons choisi de regrouper tous les contrôleurs d'assertions (qui permettent de surveiller le fonctionnement du serveur ou des différents widgets) dans un seul et même composant logiciel. Nous pouvons alors mettre en évidence les deux composants logiciels COM et MON du CDS.

Le composant COM contient les composants COM du serveur et des widgets. Il s'agit donc des composants fonctionnels originaux. Il est cependant nécessaire de leur apporter une petite modification : il faut ajouter des moyens de récupérer les observables nécessaires aux contrôleurs d'assertions. Il est donc nécessaire de mettre en place des sondes logicielles permettant de récupérer les valeurs des variables et les envois d'événements afin de les transmettre aux contrôleurs d'assertions. Ceci est symbolisé par le lien nommé Observables reliant les COM du serveur et des widgets sur la Figure 6.7.

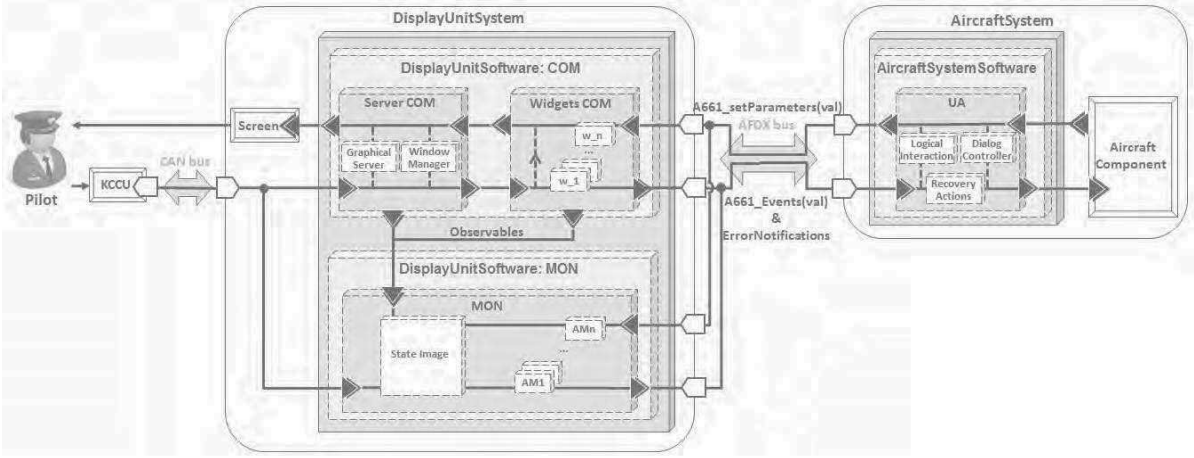


Figure 6.7. Une architecture logicielle pour la détection des erreurs

Le composant MON est constitué des différents contrôleurs d'assertions responsables de la vérification du comportement du serveur et de chaque widget. Comme nous l'avons vu lors de la description de notre méthode pour définir les contrôleurs d'assertions, ceux-ci nécessitent un stockage d'une *image de l'état* du composant COM. Cette image de l'état correspond à une version simplifiée de l'état des composants fonctionnels. Elle permet de se protéger contre la corruption de l'état du composant COM. En effet, si les contrôleurs d'assertions prennent en compte l'état du composant COM pour faire la vérification, ils ne seront pas capables de détecter les corruptions pouvant l'affecter. Il est donc nécessaire de dupliquer une partie de l'état du composant COM dans le composant MON afin de pouvoir vérifier la cohérence de l'état du composant COM (le mode de défaillance d'une fonction peut également être entrainé par une corruption de l'état sur lequel la fonction s'appuie pour s'exécuter). Le composant MON de notre architecture comprend donc une version simplifiée de l'état du composant COM, mise à jour régulièrement et que nous avons appelée State Image sur la Figure 6.7. Cette image de l'état est reconstituée par le composant MON à partir des paramètres d'entrée.

Cette image de l'état comprend notamment une version simplifiée du graphe de scène car c'est sur cette structure de données que s'appuient tous les calculs pour l'identification des widgets et le rendu graphique de l'application. Cette structure de données regroupe en effet tous les paramètres nécessaires pour faire le rendu graphique des widgets et elle permet d'effectuer le *picking* (identification du widget ciblé par l'action utilisateur) car elle est la seule à connaître l'état final des widgets nécessaire pour le calcul du picking (visibilité, activation et position). L'image de l'état est complétée par d'autres informations de l'état du composant COM telles que la position actuelle du curseur.

Nous n'intégrons pas dans notre architecture de composant de Dispatch comme il a été présenté en Figure 6.1. En effet, nous disposons ici des réseaux présents dans l'avion tels que le réseau AFDX (AEEC 2009) qui peuvent être associés avec des protocoles de diffusion fiables ; nous nous appuyons donc sur ces réseaux et leur protocole associé pour la distribution des entrées (qu'il s'agisse d'événements provenant des dispositifs d'entrée ou d'appels de méthodes provenant de l'UA) aux composants COM et MON de notre CDS autotestable. Nous nous appuyons également sur des protocoles de diffusion fiables pour la communication entre les composants COM et MON.

L'architecture proposée en Figure 6.7 présente le serveur et les widgets comme des composants autotestables, elle permet donc la détection des erreurs affectant ces composants. Il est important de proposer maintenant une solution pour le recouvrement de ces erreurs. L'UA est un excellent candidat pour le recouvrement des erreurs du CDS car il est séparé sur un autre matériel et car il a une

connaissance plus abstraite et plus sémantique du système. C'est pour cela que nous avons rajouté dans l'UA un composant logiciel nommé `Recovery Actions`. Celui-ci récupère les messages d'erreurs provenant du composant MON et décide des actions à mener en conséquence. Du fait de l'utilisation de notre système par un humain, il est possible d'intégrer celui-ci dans la boucle de recouvrement des erreurs détectées par le serveur et les widgets autotestables. Le CDS n'est pas le seul moyen pour les pilotes de commander et contrôler les systèmes avioniques d'autres mécanismes redondants et ségrégués donnent les moyens aux pilotes d'effectuer la commande et le contrôle des systèmes avioniques critiques.

Dans ce contexte, informer les pilotes de la défaillance du CDS peut être un premier pas vers le recouvrement des erreurs. En effet, si les pilotes sont informés de la défaillance du CDS, ils pourront utiliser un autre moyen pour effectuer la commande et le contrôle des systèmes avioniques critiques afin de continuer à piloter l'avion de manière sûre et sans risque d'affecter la sécurité-innocuité des passagers et de l'équipage. Cette forme de recouvrement d'erreur permet également de tenir les pilotes au courant de l'état de l'équipement qu'ils utilisent et évite les problèmes que pourraient entrainer l'automatisation du recouvrement des erreurs. Par exemple, un recouvrement automatisé pourrait introduire des comportements difficiles à analyser par les pilotes qui auraient alors une surcharge cognitive importante pour comprendre le comportement du système et continuer le pilotage de l'avion comme cela a été montré avec deux exemples de surprises d'automatisations exposés dans (Palmer 1995).

6.3.3 Architecture logicielle pour le recouvrement des erreurs

Si l'on ne souhaite pas mettre l'utilisateur au courant des erreurs du CDS, il est nécessaire de mettre en place des mécanismes pour permettre le recouvrement automatique des erreurs. Pour cela, nous proposons de nous appuyer sur l'architecture n-autotestable.

Pour tolérer les fautes, deux copies du CDS autotestable sont utilisées avec une stratégie de réplication en duplex, permettant de conserver l'état courant du système en cas d'erreur (Gibert, et al. 2012). Ainsi, les deux copies du CDS autotestable s'exécutent en parallèle. L'interaction se fait à travers l'utilisation de la première copie et l'exécution de la seconde copie est invisible pour l'utilisateur. Cela se traduit par le fait que les actions utilisateurs et les appels de méthodes de l'UA sont transmis aux deux copies mais, tant qu'il n'y a pas d'erreur, l'UA ne prend en compte que les résultats de la première copie. Si ceux-ci sont détectés défaillants, l'UA tiendra alors compte des résultats provenant de la deuxième copie.

L'avantage de cette solution réside dans le fait que le recouvrement des erreurs est transparent pour l'utilisateur. Son principal inconvénient est qu'elle demande beaucoup plus de ressources que la première architecture que nous avons présentée.

6.4 Conclusion

Nous avons présenté dans ce chapitre trois architectures tolérantes aux fautes pouvant être appliquées aux composants logiciels des systèmes interactifs : les architectures autotestables, n-autotestables et NVP. Nous avons également présenté dans quelle mesure ces architectures étaient applicables à ces différents composants et quels étaient les avantages et les inconvénients de chaque architecture.

Cette étude nous a permis de sélectionner un couple d'architectures particulièrement adaptées (autant en termes de coût de développement et de matériel qu'en termes de réalisation) pour les systèmes interactifs dans les cockpits avioniques : les architectures autotestables et n-autotestables.

Nous avons ensuite proposé d'appliquer une architecture autotestable avec contrôleur d'assertions aux systèmes interactifs présents dans les cockpits avioniques. Afin de permettre la réalisation de cette architecture, nous avons proposé une méthode pour la définition des contrôleurs d'assertions s'appuyant sur l'utilisation d'une analyse des modes de défaillance des composants logiciels concernés. Cette analyse des modes de défaillance suit la méthode des AMDEC et s'effectue à l'aide d'une description

de l'architecture logicielle du système interactif avec le langage graphique AADL et de diagrammes de séquence représentant le comportement du système.

Nous avons ensuite présenté l'architecture logicielle d'un CDS autotestable dans laquelle nous avons pu intégrer les contrôleurs d'assertions de ces différents composants logiciels. Cette architecture propose également, pour le recouvrement des erreurs, de notifier les pilotes afin que ceux-ci puissent décider d'utiliser un composant redondant au CDS pour la commande et le contrôle des systèmes critiques de l'avion. Enfin, nous avons également présenté une architecture logicielle *n*-autotestable permettant de réaliser le recouvrement automatique des erreurs dans le CDS sans impliquer les pilotes.

Chapitre 7. Mise en œuvre et validation

Sommaire

Nous avons présenté une architecture logicielle tolérante aux fautes pour les systèmes interactifs dans les cockpits avioniques (voir Chapitre 6) ainsi qu'une approche basée sur les modèles pour la conception, la spécification et le développement des systèmes interactifs permettant une conception des systèmes interactifs au plus proche du zéro-défaut (voir Chapitre 5). Nous proposons dans ce chapitre des solutions et des outils pour la mise en œuvre de ces deux contributions.

La première section présente l'outil PetShop qui permet la modélisation, l'exécution et l'analyse des modèles ICO.

La deuxième section propose une maquette pour la validation de notre architecture logicielle. Cette maquette s'appuie sur l'utilisation d'ARISSIM, un simulateur de noyau avionique respectant le standard ARINC 653.

Enfin, la troisième section présente des pistes pour la validation de notre approche qui permet de prévenir les défaillances des systèmes interactifs. Ces défaillances peuvent être dues à des fautes logicielles de développement, des fautes matérielles en opération ou des fautes liées à des problèmes d'utilisabilité. Ainsi, nous présentons trois ébauches différentes pour la validation de ces trois aspects.

7.1 PetShop : un outil pour la mise en œuvre de notre approche basée sur les modèles

7.1.1 Les besoins d'un outil pour supporter la modélisation avec la notation ICO

Notre méthode pour concevoir des systèmes interactifs zéro-défaut, présentée dans le Chapitre 5, s'appuie sur la modélisation des composants logiciels du système interactif à l'aide de la technique de description formelle ICO (D. Navarre, P. Palanque, et al. 2009) et (Hamon 2014).

Ainsi, le comportement des différents composants du système interactif est décrit par des modèles ICO. Un modèle ICO est constitué d'un Objet Coopératif (CO) à qui il a été associé une fonction de rendu et une fonction d'activation. Un Objet Coopératif est une instance d'une CO-classe qui

implémente une interface Java et dont le comportement est décrit par un réseau de Petri haut-niveau appelé ObCS (pour Structure de Contrôle de l'Objet, Object Control Structure).

Pour rendre notre méthode et la notation ICO utilisables, il est nécessaire de proposer des outils pour supporter la modélisation de systèmes interactifs à l'aide de cette technique de description formelle (Navarre 2001). En effet, les systèmes interactifs sont complexes et nécessitent la modélisation de nombreux composants (voir Chapitre 5). De plus, les modèles ICO de ces composants présentent un grand nombre d'informations, il est donc difficile de développer de tels modèles sans un environnement logiciel.

Cet environnement de conception a pour but de modéliser et spécifier des systèmes interactifs. Ainsi, il nécessite un certain nombre de fonctionnalités :

- Il doit permettre l'édition de modèles ICO. Pour cela, il doit permettre à la fois l'édition des CO (et donc de leur interface en Java et leur ObCS) mais également des fonctions de rendu et d'activation et du rendu graphique final du système interactif.
- Il doit permettre l'analyse statique des modèles ICO (voir section 7.3.1).
- Il doit supporter le passage à l'échelle pour les modèles compliqués permettant de spécifier des études de cas réalistes telle que celle que nous présentons dans la Partie 3.
- Il doit permettre de gérer plusieurs modèles ainsi que leurs connexions.
- Il doit permettre de simuler les modèles pour permettre l'évaluation des systèmes spécifiés et vérifier que le comportement spécifié correspond au comportement attendu.
- Il doit permettre d'enregistrer tous les événements se produisant pendant l'utilisation du système (actions utilisateurs, franchissement de transitions dans les modèles ICO, les jetons entrants ou sortants des places…) afin de générer des fichiers de journalisation (*log*) qui permettront par exemple l'analyse du système d'un point de vue efficacité (Palanque, Barboni, et al. 2011).
- Il doit également être suffisamment utilisable comme montré par les travaux de (Barboni, Bastide, et al. 2003).

7.1.2 Présentation générale

Nous décrivons dans cette section l'environnement logiciel PetShop qui permet la mise en œuvre de notre méthode et qui correspond à tous les critères énoncés dans la section précédente.

L'outil PetShop (pour Petri Net workShop (Barboni, Ladry, et al. 2010)) est un environnement d'édition, de vérification et d'exécution des modèles ICO. Nous reprenons ici la présentation de l'outil PetShop dans sa dernière version qui a été réalisée dans les travaux de (Hamon 2014). Cet outil est développé par l'équipe de recherche en systèmes interactifs critiques (ICS[2]) de l'IRIT (Institut de Recherche en Informatique de Toulouse).

La Figure 7.1 présente une copie d'écran de PetShop. L'agencement de l'espace de travail est personnalisable en fonction de l'activité du développeur. Cet agencement présente les vues suivantes :

1. *Un navigateur de projets* qui contient les modèles ICO, les interfaces Java de leurs CO-classes ainsi que les classes Java pour le rendu graphique du système.
2. *Une fenêtre pour l'édition de modèles ICO* qui permet également leur visualisation et leur analyse.
3. *Une palette d'édition pour les modèles ICO* qui permet de glisser/déposer les différents éléments de la notation dans le modèle.
4. *Un éditeur de code Java* qui permet d'éditer les interfaces de CO-classes ainsi que les classes de rendu graphique du système.
5. *Un débogueur*.

[2] Équipe de recherche ICS : http://www.irit.fr/recherches/ICS/ (au 1er mars 2015)

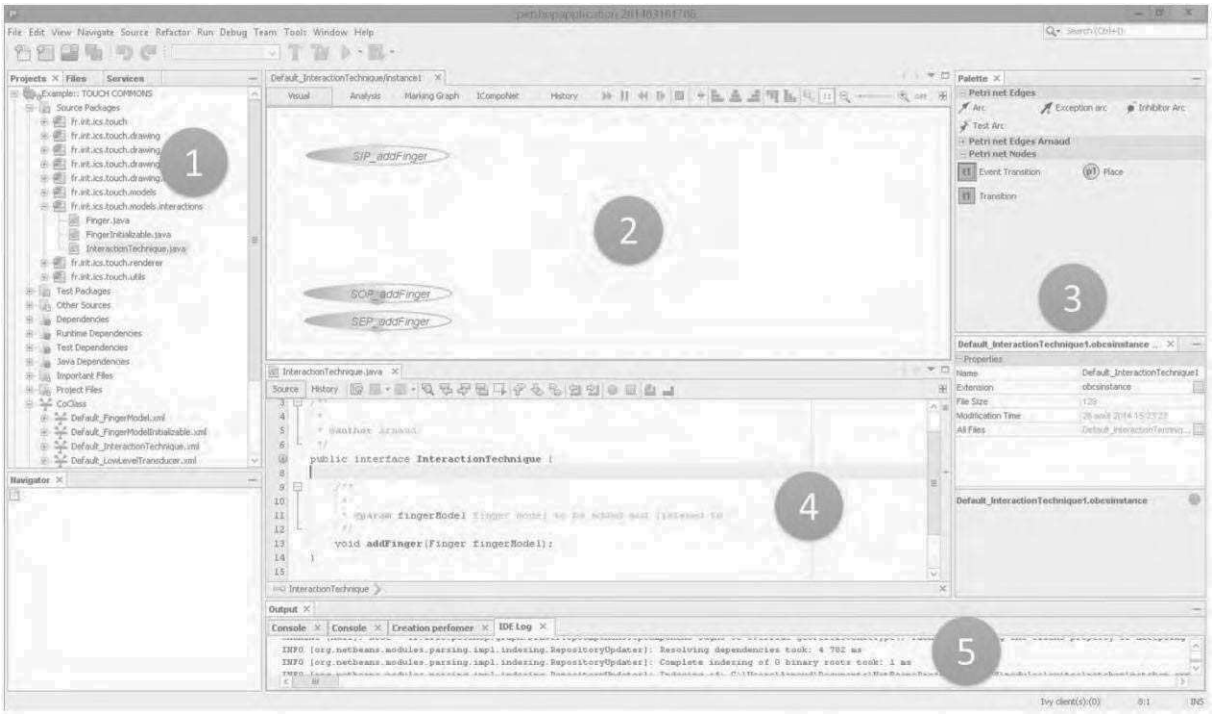


Figure 7.1. Copie d'écran de l'environnement de PetShop (Hamon 2014)

7.1.3 Principes de fonctionnement

L'outil PetShop est développé avec le langage Java, ce qui rend possible son déploiement sur les systèmes d'exploitation les plus répandus tels que Windows, Linux ou MacOs.

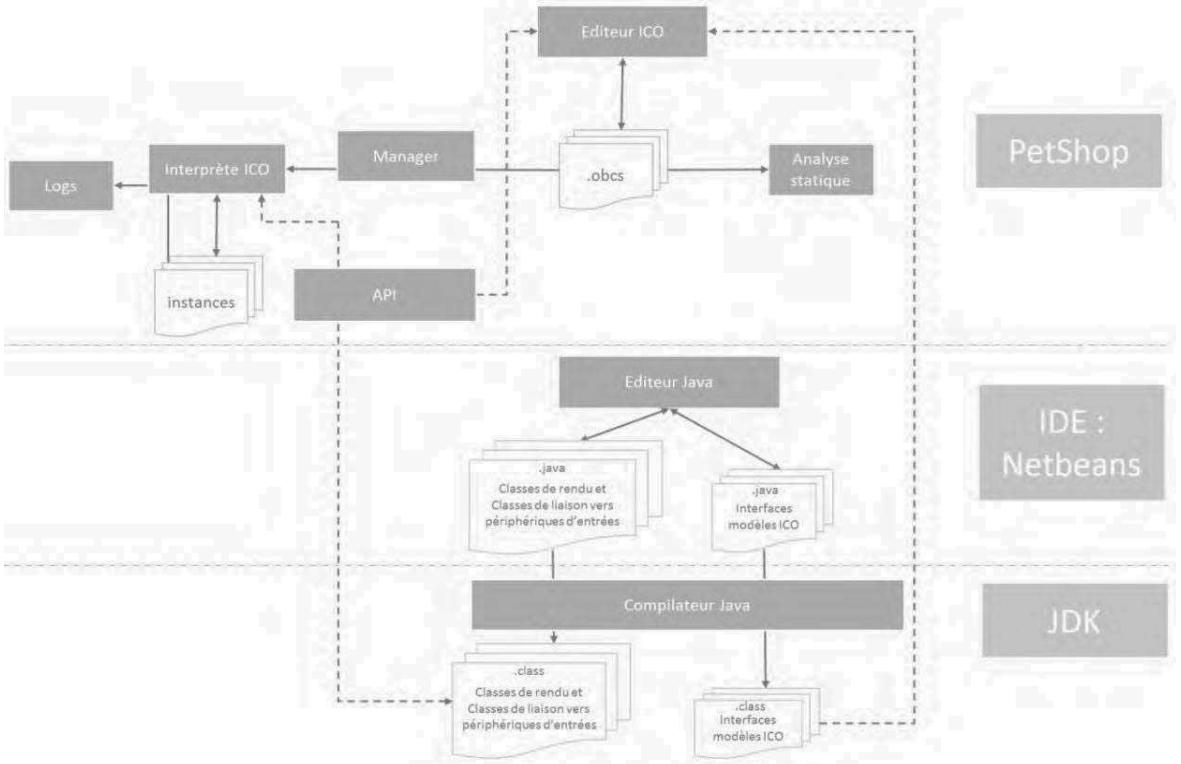


Figure 7.2. Schéma illustrant les principes de fonctionnement de PetShop (Hamon 2014)

Les principes de fonctionnement de l'outil PetShop sont représentés sur la Figure 7.2. Ainsi, la dernière version de PetShop est une application NetBeans [3] . PetShop s'appuie sur certaines fonctionnalités de cet IDE pour intégrer un éditeur et un compilateur de code Java ainsi qu'un gestionnaire de projets et de versions qui facilitent grandement le travail en équipe et l'édition des classes Java associées aux modèles ICO.

L'éditeur de modèles ICO utilise le résultat de la compilation des interfaces Java des CO-classes pour créer la structure des ICO. Les modèles produits sont sauvegardés dans des fichiers ObCS.

L'interprète permet de simuler les modèles ICO tout en les liant avec les classes Java qui décrivent à la fois le rendu graphique du système interactif et les liens entre les modèles ICO et les périphériques d'entrée et de sortie. Il assure également la communication entre les différents modèles ICO. Enfin l'interprète assure l'enregistrement de tous les événements des différents modèles ICO pour les sauvegarder dans les fichiers de journalisation (*log*).

Nous développons dans les sous-sections suivantes les différentes fonctionnalités de l'outil PetShop.

7.1.4 Édition, interprétation et débogage des modèles ICO

Afin de permettre le prototypage du système interactif, PetShop ne propose pas de séparation entre les activités d'édition et d'exécution des ICO (Navarre 2001). Ainsi, les modèles ICO sont en permanence édités, exécutés et analysés. On peut donc, par exemple, rajouter des places ou des transitions durant l'exécution des modèles ICO afin de modifier en « temps réel » le comportement du système interactif.

L'outil permet de modéliser tous les éléments des modèles ICO en parallèle et au sein d'un même projet. Ainsi, il permet l'édition des CO (leur interface en Java et leur ObCS, autrement dit leur comportement en réseaux de Petri haut-niveau). Il permet également l'édition des fonctions de rendu et d'activation ainsi que du rendu graphique final du système interactif et des liens avec les périphériques d'entrée et de sortie.

Pour chaque modèle, PetShop fournit également sa représentation en CompoNet (Barboni 2006) qui présente les différents ports de communication d'une CO-classe. La Figure 7.3 présente la représentation dans PetShop du modèle CompoNet générique d'un widget.

Figure 7.3. Visualisation du modèle CompoNet générique d'un widget dans PetShop

Figure 7.4. Copie d'écran du contrôleur d'instance de PetShop

Enfin, PetShop fournit, pour chaque instance de modèle, un moyen de contrôler son exécution à travers un contrôleur d'instance semblable à une commande de magnétoscope (voir Figure 7.4). Par défaut, l'exécution est en mode automatique. Cependant, l'utilisateur peut choisir de suspendre temporairement cette exécution. Il pourra alors exécuter l'interprétation du modèle ICO en mode manuel et évaluer pas à pas le franchissement des transitions en choisissant les substitutions valides qui seront utilisées.

[3] NetBeans : https://netbeans.org/ (au 12 mai 2015)

7.1.5 L'analyse des modèles ICO

L'outil PetShop propose également un outil d'analyse mathématique et statique des modèles ICO. La Figure 7.5 présente une copie d'écran de l'analyse d'un modèle ICO. On peut ainsi retrouver :

1. La matrice d'incidence du réseau.
2. Les invariants de place.
3. Les invariants de transition.
4. Les places puits.
5. Les places sources.

Cette analyse est disponible dès la création du modèle ICO. Elle ne nécessite pas de compilation et est mise à jour à chaque modification du modèle (et ce, même lors de l'exécution de celui-ci).

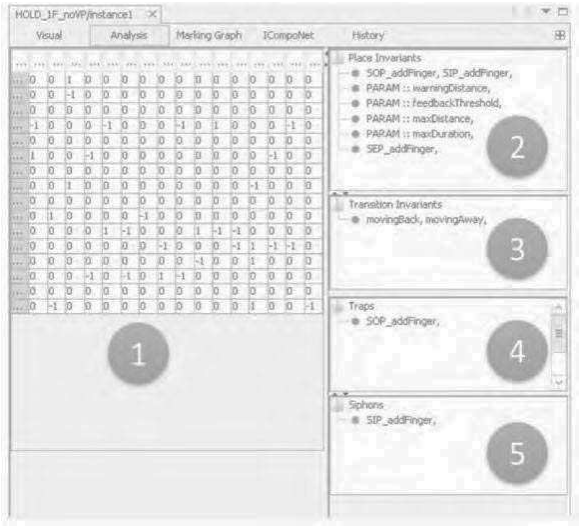


Figure 7.5. Copie d'écran du résultat du module d'analyse des modèles ICO dans PetShop (Hamon 2014)

7.2 Mise en œuvre de notre architecture logicielle

7.2.1 Mise en œuvre de notre architecture logicielle autotestable

Pour que notre architecture logicielle soit efficace, il est important de ségréguer les différents composants. En effet, sans ségrégation, il n'y a aucune garantie quant au confinement des erreurs. C'est-à-dire, il n'y a aucune garantie qu'une faute provoquant une erreur dans l'un des composants ne crée pas également une erreur dans un autre composant.

Pour réaliser la ségrégation spatiale et temporelle de nos composants, nous proposons de nous appuyer sur les concepts définis par le standard ARINC 653 (AEEC 2003) qui est le standard de référence pour la ségrégation spatiale et temporelle dans l'avionique.

Dans le cas de notre architecture, il s'agit de ségréguer les deux composants constituant le système interactif autotestable (les composants `COM` et `MON` du CDS). Afin d'augmenter le confinement d'erreur et de pouvoir détecter les défaillances dues à un arrêt du système (qui pourrait être causé par une coupure d'alimentation par exemple), nous séparons les deux composants `COM` et `MON` sur deux processeurs différents, autrement dit, deux DU (Display Unit) différentes.

Cette architecture est représentée par la Figure 7.6 sur laquelle nous retrouvons deux applications interactives : l'une critique qui implémente notre architecture autotestable (`appli1`) et la seconde non critique n'implémentant pas l'architecture autotestable (`appli2`). La `DU1` est consacrée à l'affichage (et donc à l'exécution du composant `COM`) de l'`appli1` alors que la `DU2` est consacrée à l'affichage de l'`appli2`. La partition `MON appli1` vérifie le comportement de la partition `COM appli1`.

Toutes les entrées concernant une même application sont redirigées vers les composants COM et MON de cette application à l'aide d'un protocole de diffusion fiable. Les entrées sont traitées par le composant COM et mises de côté par le composant MON. Celui-ci les traite avec un temps de cycle de retard par rapport au traitement du composant COM. En effet, la vérification de la validité des résultats du composant COM, implique de devoir attendre que ceux-ci soient délivrés, d'où ce décalage. Les UA reçoivent les résultats du composant COM et les traitent lors de la réception de leur validation par le composant MON.

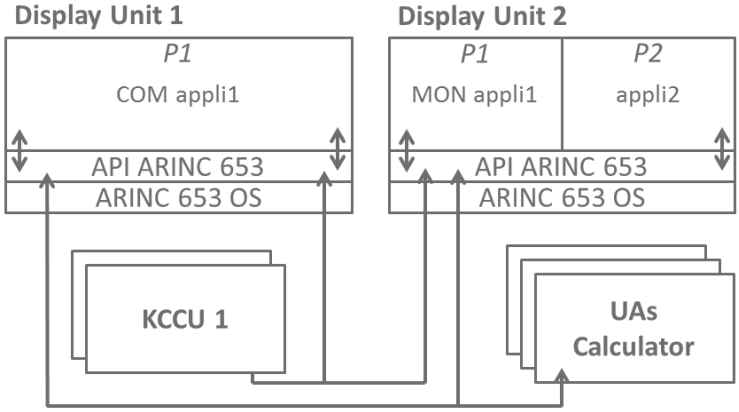


Figure 7.6. Architecture physique et partitionnement spatial pour la mise en œuvre des systèmes interactifs autotestables

La Figure 7.7 présente le partitionnement temporel des deux DU représentées en Figure 7.6. Nous pouvons ainsi mettre en évidence le temps alloué à chaque partition (t1 pour la partition MON appli1 et t2-t1 pour la partition appli2 ; la partition COM appli1 est la seule à s'exécuter sur la DU1) par rapport au temps de cycle d'une DU. Ce temps de cycle doit être inférieur à la moitié du temps moyen que met un humain pour percevoir une information, de manière à pouvoir garantir une fluidité de l'affichage pour l'utilisateur et ne pas perturber l'utilisabilité du système. Le temps moyen de perception pour un humain étant de 70 ms (Card, Newell et Moran 1983), nous utilisons un temps de cycle de 33ms qui est le temps de cycle utilisé à l'heure actuelle dans les DU des Airbus A380.

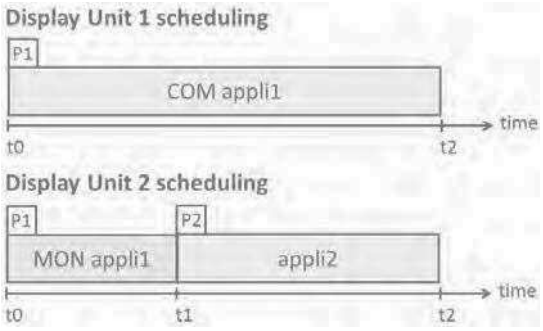


Figure 7.7. Partitionnement temporel pour la mise en œuvre des systèmes interactifs autotestables

Il est important de noter que le partitionnement temporel et spatial proposé implique des problèmes de synchronisation qu'il faudra prendre en compte. Ainsi, étant donné que les deux DU n'ont pas une horloge commune, les partitions ne s'exécutent pas en parallèle comme présenté sur la Figure 7.7 et vont s'exécuter en décalé. En pratique, la dérive temporelle entre les deux horloges est bornée à un temps de cycle. Une étude plus poussée sur ces aspects de synchronisation peut être trouvée dans (Lauer, et al. 2011). De plus, la vérification des résultats du composant COM par le composant MON implique un retard d'au minimum un temps de cycle car le composant MON nécessite d'attendre que le composant COM ait traité les entrées pour pouvoir les vérifier. Il est important de s'assurer que ce retard reste minimal car autrement, il pourrait affecter l'utilisabilité du système et gêner les pilotes dans leur travail.

7.2.2 Mise en œuvre de notre architecture logicielle n-autotestable

L'architecture logicielle *n*-autotestable diffère de l'architecture autotestable de par le fait de la réplication du composant autotestable. Nous avons proposé dans le chapitre précédent une architecture

2-autotestable qui a l'avantage, par rapport à l'architecture autotestable, de rendre possible le recouvrement automatique des erreurs.

L'architecture matérielle associée à cette architecture logicielle est présentée en Figure 7.8. L'application appli1 implémente l'architecture 2-autotestable et est donc composée de deux composants COM et deux composants MON ségrégués sur deux DU. Ainsi, une première copie du CDS autotestable (pour l'`appli1`) est constituée des composants `COM1` et `MON1`, respectivement situés dans les partitions `P2` de la `DU1` et `P1` de la `DU2`. La deuxième copie du CDS autotestable est constituée des composants `COM2` et `MON2`, respectivement situés dans les partitions `P2` de la `DU2` et `P1` de la `DU1`. L'application `appli2` n'étant pas critique, elle n'instancie pas l'architecture 2-autotestable est s'exécute dans une troisième partition de la `DU2`.

L'utilisateur interagit avec la première copie du CDS autotestable qui s'exécute en parallèle de la deuxième. L'UA ne tient pas compte des résultats envoyés par la deuxième copie tant que ceux de la première copie ne comportent pas d'erreur. Les entrées provenant des actions de l'utilisateur sur le KCCU et celles provenant de l'UA sont transmises aux deux composants `COM` et aux deux composants `MON` grâce à un protocole de diffusion fiable. En cas d'erreur sur la première version du CDS autotestable (`COM1` et `MON1`), l'`appli1` s'exécute sur la `DU2` et l'`appli2` n'est plus accessible à l'utilisateur.

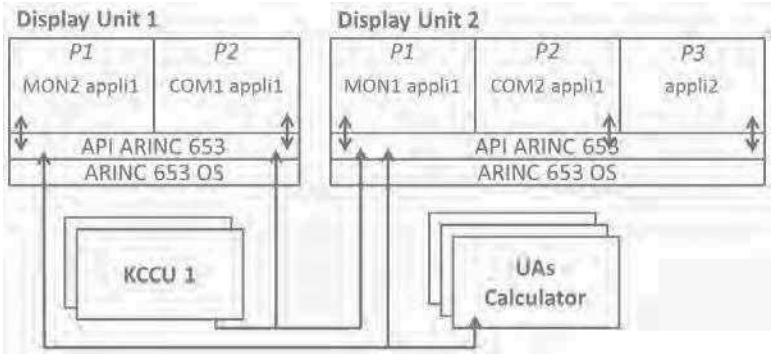


Figure 7.8. Architecture physique et partitionnement spatial pour la mise en œuvre des systèmes interactifs n-autotestables

Présentée telle quelle, cette solution peut être vue comme de la reconfiguration. Cependant, elle pourrait être transparente à l'utilisateur comme nous l'avons expliqué dans la section 6.3.3 si l'on considérait des DU contenant des processeurs multi-cœur. En effet, dans ce cas, les deux copies du CDS autotestables pourraient être exécutées chacune sur un cœur différent, ce qui permettrait de faire la reconfiguration en cas d'erreur sur la même DU et donc sur le même écran. Cette configuration permettrait de rendre le recouvrement d'un certain nombre d'erreurs complètement transparent pour l'utilisateur.

7.2.3 ARISSIM : un simulateur respectant le standard ARINC 653

7.2.3.1 Les besoins d'un simulateur de système d'exploitation avionique

Nous cherchons à réaliser nos architectures logicielles et matérielles de manière à ce que leur mise en œuvre soit la plus représentative de celle qui pourrait être faite dans un cockpit d'avion afin de valider nos architectures. Il est donc nécessaire de mettre en œuvre nos architectures logicielles sur un système permettant de respecter la ségrégation spatiale et temporelle du standard ARINC 653.

Plusieurs systèmes d'exploitation respectant ce standard sont disponibles. Nous pouvons ainsi citer, pour les systèmes d'exploitation commerciaux, *VxWorks*[4] de Wind River, *LynxOS*[5] de Lynx Software

[4] VxWorks : http://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653 (au 12 mai 2015)

[5] LynxOS : http://www.lynx.com/products/real-time-operating-systems/lynxos-178-rtos-for-do-178b-software-certification/ (au 12 mai 2015)

Technologies, *INTEGRITY-178 RTOS*[6] de Green Hills Software et *PikeOS*[7] de Sysgo ; l'unique solution open source que nous avons pu trouver est le système d'exploitation POK[8], qui a permis de supporter plusieurs travaux de recherche et permet également de développer le code des partitions à partir d'une spécification AADL. Les systèmes commerciaux sont extrêmement coûteux et ne sont pas accessibles dans un but de recherche. De plus, il serait également très coûteux en temps de développement de s'adapter à un système d'exploitation, même lorsque l'on considère POK qui pourrait être accessible, il faudrait en effet coder les différents composants du système interactif tolérant aux fautes en langage C et l'adapter aux contraintes de ces systèmes d'exploitation. Ces différents systèmes ne supportent également pas le langage Java, ce qui nous empêcherait d'utiliser PetShop pour l'exécution de nos modèles ICO.

Il n'est donc pas intéressant dans notre cas d'utiliser un vrai système d'exploitation respectant le standard ARINC 653, nous nous sommes donc intéressés aux simulateurs existants car ceux-ci permettent de moins fortes contraintes de développement et sont donc beaucoup plus adaptés à des objectifs de test et de preuve de concept de notre mise en œuvre. Ce genre de système a beaucoup été étudié par la recherche mais très peu de simulateurs de ce genre sont encore une fois accessibles. Nous pouvons ainsi citer le simulateur AMOBA (Pascoal, et al. 2008), intégré au simulateur SIMA[9] qui permet de simuler des plateformes avioniques modulaires intégrées (ou IMA voir section 3.2.3) (Santos, et al. 2008) et (Schoofs, et al. 2009). Ce simulateur est accessible de manière commerciale. Il existe également ACM (Dubey, Karsai et Kereskenyi, et al. 2010), un modèle composant du standard ARINC 653, auquel est associée une suite d'outils appelée ACMTOOLSUITE[10] (Dubey, Karsai et Mahadevan, A Component Model for Hard Real-time Systems: CCM with ARINC-653 2011) ; mais encore une fois, celui-ci ne permet pas l'utilisation du langage Java pour le développement du code des partitions.

Nous avons donc choisi de développer notre propre simulateur de système d'exploitation respectant le standard ARINC 653 (Bustamante et Palustran 2013) et (Cronel 2013) que nous décrivons dans la section suivante.

7.2.3.2 ARISSIM : un simulateur de système d'exploitation avionique respectant le standard ARINC 653

Fonctionnalités du simulateur

L'intérêt premier de l'utilisation d'un simulateur de système d'exploitation respectant le standard ARINC 653 est, dans notre cas, de pouvoir s'appuyer sur ses principes de partitionnement qui permettent une ségrégation spatiale et temporelle entre les différentes partitions. Ce partitionnement garanti qu'il ne peut pas y avoir de propagation de faute entre les différents composants de notre architecture logicielle. Notre simulateur respecte donc la ségrégation spatiale et temporelle entre les différentes partitions.

Dans un second temps, le simulateur doit permettre la communication entre les différentes partitions. En effet, il est indispensable de pouvoir garantir la communication entre les différents composants de l'architecture logicielle, et plus particulièrement celle entre les composants `COM` et les composants `MON`. Du fait des applications auxquelles elles sont destinées et de leur complémentarité, nous avons choisi d'implémenter les deux types de communication inter-partitions décrits par le standard ARINC 653 : la communication par sampling (adaptée pour l'envoi de messages où seule la dernière valeur est importante) et la communication par queuing (adaptée pour l'envoi de messages multiples dans le cas où ils sont tous importants).

---

[6] INTEGRITY-178 RTOS : http://www.ghs.com/products/safety_critical/integrity-do-178b.html (au 12 mai 2015)

[7] PikeOS : http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/safety-architecture/ (au 12 mai 2015)

[8] POK : http://pok.tuxfamily.org/ (au 12 mai 2015)

[9] SIMA : http://www.gmv.com/en/aeronautics/products/sima/index.html (au 12 mai 2015)

[10] ACMTOOLSUITE : https://wiki.isis.vanderbilt.edu/mbshm/index.php/ACMTOOLSUITE (au 12 mai 2015)

Afin de pouvoir simuler un cockpit avionique plus complet et de permettre la mise en œuvre de l'architecture *n*-autotestable, il est également nécessaire que le simulateur puisse fonctionner sur plusieurs machines différentes, plus exactement, il est important de garantir la communication entre plusieurs instances du simulateur sur des machines différentes.

Enfin, pour permettre l'utilisation de l'outil PetShop et ainsi permettre l'interprétation directe des modèles du système interactif, il est important que le code implémenté dans les partitions du simulateur puisse être du Java.

Description des principes de fonctionnement du simulateur ARISSIM

ARISSIM [11] (qui signifie **ARI**NC 653 **S**tandard **SIM**ulator) est un simulateur de système d'exploitation respectant les principes de communication et de partitionnement spatial et temporel du standard ARINC 653. Ce simulateur a été développé au sein du groupe TSF du LAAS-CNRS[12] afin de pouvoir servir à des objectifs de recherche et d'enseignement.

La Figure 7.9 illustre le principe de fonctionnement du simulateur ARISSIM. Celui-ci a été développé de manière à fonctionner sur les systèmes d'exploitation Unix. Le paramétrage du simulateur est défini par deux fichiers de configuration. Le premier fichier de configuration permet de définir les différentes partitions qui doivent être exécutées par le simulateur ainsi que la période d'exécution qui leur est affectée. Le second fichier de configuration permet de définir les différents canaux de communication entre les partitions.

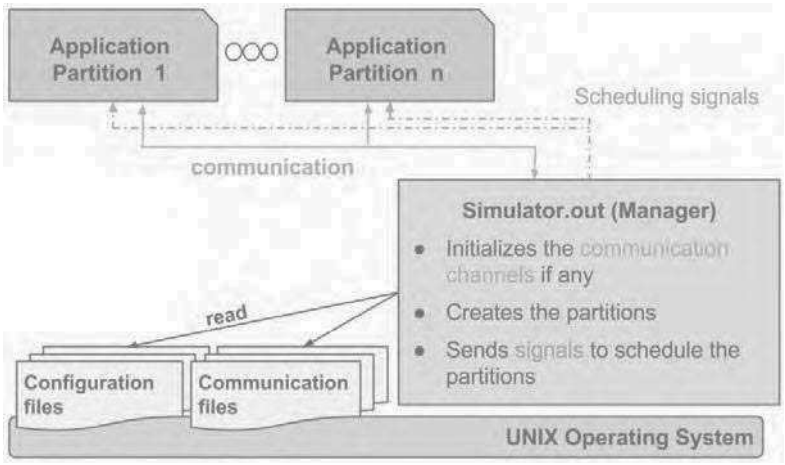


Figure 7.9. Principe de fonctionnement du simulateur ARISSIM (Cronel 2013)

Chaque partition correspond à un processus Unix multitâche. Le partitionnement spatial, c'est-à-dire l'isolation entre les zones de mémoire affectée à chaque partition, s'appuie sur la notion de processus et les protections mémoire associées du système d'exploitation Unix. Celle-ci permet la création d'un nouveau processus complètement indépendant et garantit l'isolation de la mémoire. Ainsi, les partitions ne partagent aucun espace mémoire.

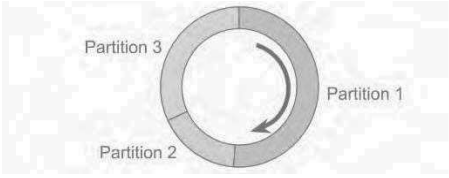


Figure 7.10. Ordonnancement temporel du simulateur (Cronel 2013)

Le partitionnement temporel, illustré en Figure 7.10, est fixe durant l'exécution des partitions (il est fixé par le premier fichier de configuration). Il s'appuie sur l'utilisation des signaux Unix (`SIGSTOP`

[11] ARISSIM : http://makrin.github.io/ARISS/ (au 12 mai 2015)

[12] Groupe TSF du LAAS-CNRS : https://www.laas.fr/public/en/tsf (au 12 mai 2015)

et SIGCONT) qui permettent l'ordonnancement des processus Unix. Ces deux signaux sont envoyés aux différentes partitions et permettent de stopper ou d'activer leur exécution. Les différentes partitions sont ainsi exécutées dans un ordre cyclique et ne partagent aucun temps d'exécution commun.

La communication inter-partition s'appuie sur l'utilisation des sockets Unix. La communication par *queuing* est assurée par une communication point à point s'appuyant sur la communication par sockets qui permet ainsi de garantir une communication par file d'attente sans aucune perte de message. La communication par *sampling* est assurée également grâce à l'utilisation des sockets mais est réalisée de manière à ce que seul le dernier message soit lisible et que la lecture ne supprime pas celui-ci. L'utilisation des sockets permet également une communication entre différentes machines sur lesquelles serait exécuté un simulateur ARISSIM en utilisant le réseau internet. Pour leur permettre d'utiliser les canaux de communications, le simulateur fournit aux partitions une librairie de fonctions permettant l'envoi et la réception de messages pour les deux modes (queuing et sampling).

Le code source du simulateur est disponible en libre accès depuis octobre 2014. Il a également été utilisé pour plusieurs projets étudiants en partenariat avec Airbus Defence and Space, le LAAS-CNRS et l'INP-ENSEEIHT[13] où il a servi pour le développement d'une maquette de système embarqué satellite AGILE de prise de vue sur une carte Raspberry Pi (Beaussart, et al. 2014) ainsi que pour l'analyse de sûreté de fonctionnement de cette maquette (Bedoin, et al. 2015).

7.2.4 Architecture matérielle et logicielle de la maquette

Nous proposons de faire la preuve de concept de notre architecture logicielle autotestable à l'aide d'une maquette qui nous permettra de valider les concepts développés dans cette thèse. L'architecture logicielle et matérielle de cette maquette est présentée en Figure 7.11.

Elle est développée sur un réseau de trois ordinateurs sous Linux sur lesquels tourne le simulateur ARISSIM. Le premier ordinateur (nommé PC1) exécute le code correspondant au composant COM du CDS dans une partition P1. Le deuxième ordinateur (nommé PC2) exécute le code correspondant au composant MON du CDS dans une partition P1. Pour finir, le troisième ordinateur (nommé PC3) exécute le code correspondant au composant UA associé, encore une fois dans une partition P1.

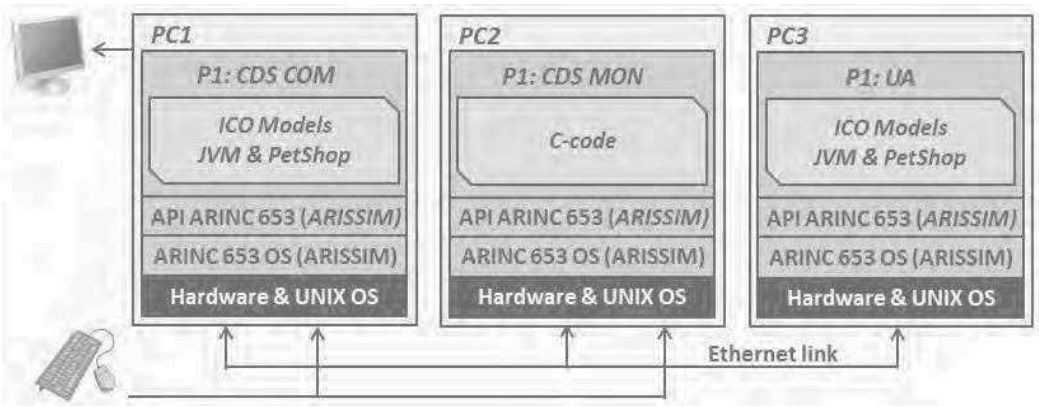


Figure 7.11. Architecture logicielle et matérielle de la maquette de test des systèmes interactifs tolérants aux fautes

Afin de simplifier la lecture, nous ne présenterons pas ici la maquette correspondante à l'architecture logicielle *n*-autotestable. Cependant, la copie du composant COM pourrait être exécutée sur une partition P2 du PC2 et la copie du composant MON pourrait être exécutée sur une partition P2 du PC1.

Le composant COM est développé en utilisant notre approche à base de modèles (voir Chapitre 5). Les différents composants logiciels du CDS sont donc modélisés à l'aide du formalisme ICO et

[13] Chaire CESEC : http://websites.isae.fr/cesec-chair (au 12 mai 2015)

respectent le standard ARINC 661 (AEEC 2013). Tous les modèles sont exécutés à l'aide de l'outil PetShop qui s'exécute sur une machine virtuelle Java.

Le composant MON est implanté en langage C afin de le diversifier au maximum vis-à-vis du composant COM et de pouvoir couvrir les fautes qui peuvent affecter le support d'exécution des modèles ICO (l'outil PetShop et la machine virtuelle Java).

La communication entre les composants COM et MON est garantie par deux canaux de communication, le premier utilisant le mode de communication queuing et le second utilisant le mode de communication sampling. Toutes les informations nécessaires au composant MON pour effectuer la vérification des assertions sont transmises au travers de ces deux canaux. Le canal de communication en mode queuing est utilisé pour transmettre les événements (tels que les événements envoyés par les widgets à l'UA) et le canal de communication en mode sampling est utilisé pour transmettre des valeurs de variables (telles que les paramètres constituant l'état d'un widget). Les sondes permettant de transmettre les informations provenant du composant COM et nécessaires au composant MON pour effectuer le contrôle des assertions observent les modèles ICO en utilisant les fonctionnalités fournies par l'outil PetShop tel que la possibilité d'accéder à la valeur d'un jeton au travers l'accès à l'interprète des modèle ICO.

7.3 Pistes de validations de notre approche

L'approche proposée dans la thèse permet de prévenir les défaillances des systèmes interactifs. Ces défaillances peuvent être dues à trois types de fautes différentes :

- Les fautes logicielles de développement.
- Les fautes matérielles en opération.
- Les fautes liées à des problèmes d'utilisabilité.

Pour valider notre approche par rapport à la prévention des défaillances liées à ces trois types de fautes, nous présentons dans cette section trois ébauches différentes pour la validation de ces trois aspects. Premièrement, pour valider notre approche à base de modèles pour la prévention des défaillances dues aux fautes logicielles de développement, nous présentons des moyens d'analyse formelle sur les modèles ICO permettant de supporter les activités de vérification et de validation du logiciel. Deuxièmement, pour valider notre approche pour la prévention des défaillances dues aux fautes naturelles en opération (architecture logicielle tolérante aux fautes), nous présentons une ébauche d'approche pour effectuer une campagne d'injection de fautes afin de valider les mécanismes d'injection de fautes. Troisièmement, pour valider nos deux contributions par rapport à la prévention des défaillances liées aux problèmes d'utilisabilité, nous présentons une approche pour vérifier que les mécanismes mis en place n'affectent pas l'utilisabilité du système.

7.3.1 Analyse formelle des modèles ICO

7.3.1.1 Généralités

Le choix de la notation ICO pour la description comportementale des systèmes interactifs a été en partie motivé par le fait que celle-ci permette de pratiquer des analyses formelles sur les modèles permettant de servir les activités de vérification et de validation du logiciel. Ces deux activités sont en effet fondamentales pour le développement d'un logiciel zéro-défaut car elles permettent de confirmer que le système développé respecte les exigences spécifiées et qu'il permet de fournir le service désiré.

Nous ne traitons pas ces deux activités dans ce document car elles s'intègrent dans un processus de développement complet, ce qui est hors du périmètre de cette thèse. Cependant, ces deux activités sont considérées lorsque l'on applique des processus tels que celui recommandé la norme DO-178C (RTCA et EUROCAE 2011a) ou tels que celui proposé par les travaux de (Martinie, Palanque et Navarre, et al. 2012) qui permet d'intégrer les aspects sûreté de fonctionnement et utilisabilité pour les systèmes interactifs.

La notation ICO permet d'effectuer différents types d'analyse formelle. Premièrement, le fait qu'elle soit fondée sur les réseaux de Petri implique qu'elle rende possible plusieurs types d'analyses formelles sur la structure des modèles ainsi que sur les graphes de marquage des réseaux. Deuxièmement, la notation ICO permet également d'effectuer du model-checking à travers l'utilisation d'un outil spécialisé. Nous présentons dans les sous-sections suivantes les différents moyens d'analyse formelle supportés par la notation ICO ainsi que dans quelle mesure ils peuvent servir les activités de vérification et de validation du logiciel.

Il est important de noter que la vérification et la validation séparée de chaque modèle n'est pas suffisante et qu'il est nécessaire d'étudier les propriété d'offres et de demandes entre les différents modèles afin de s'assurer de la cohérence entre tous les modèles permettant de d'écrire le système interactif complet. Ceci peut être effectué en utilisant la spécification formelle des appels de méthodes dans la notation ICO. La spécification formelle des événements reste encore à étudier.

7.3.1.2 Analyse sur les réseaux de Petri sous-jacents aux modèles ICO

Une première sorte d'analyse formelle pouvant être pratiquée avec la notation ICO correspond aux analyses qui peuvent être effectuées sur les réseaux de Petri sous-jacents aux modèles ICO. Elles peuvent être des analyses sur la structure des modèles ou bien encore sur les graphes de marquage.

Ces analyses permettent de vérifier un certain nombre de propriétés sur les modèles. Afin d'illustrer la manière dont ces analyses peuvent être utilisées pour vérifier les propriétés sur les modèles ICO, nous présentons ci-dessous un exemple d'analyse des invariants de place et de transitions sur le modèle du contrôleur de dialogue de l'application « les 4 saisons » qui nous permettent de vérifier qu'une action sera toujours disponible pour l'utilisateur et que le marquage de ce modèle correspondra toujours à un état correct de l'application.

Exemple d'analyse sur la structure des modèles : analyse des invariants de places et de transitions

L'*analyse des invariants de places* (P-invariant) permet de vérifier qu'un ensemble de places contiendra toujours un jeton, c'est-à-dire qu'aucun jeton ne sera perdu lors du passage dans ces places. Concrètement, cela signifie que, quel que soit l'état du réseau, ces places ne perdront jamais leur ressource, c'est-à-dire, que le nombre de jeton initial dans ce groupe de place restera inchangé.

L'*analyse des invariants de transitions* (T-invariant) permet de vérifier la vivacité des transitions. Une transition vivante sera toujours franchissable à un moment donné, quel que soit l'état initial du réseau. Concrètement, cette analyse permet de vérifier qu'un service sera toujours disponible pour l'utilisateur.

Prenons l'exemple du contrôleur de dialogue de l'application « les 4 saisons ». Le modèle ICO de ce composant est rappelé en Figure 7.12. Le résultat de l'analyse des invariants sur le réseau de Petri sous-jacent à ce modèle est présenté en Figure 7.13. Nous pouvons constater que ce modèle contient un invariant de place composé des places {`spring`, `summer`, `fall`, `winter`} ainsi qu'un invariant de transitions composé des transitions {`t1`, `t2`, `t3`, `t4`}.

L'invariant de place nous permet de vérifier que le nombre de jetons présents, pendant l'état initial, dans l'ensemble de ces quatre places ne variera pas lors de l'exécution. Si nous considérons l'exemple de la Figure 7.12 comme étant l'état initial, l'invariant de place {`spring`, `summer`, `fall`, `winter`} nous permet de garantir qu'un jeton sera toujours présent dans l'une de ces quatre places. Plus concrètement, étant donné que l'état du système est représenté par la présence d'un jeton dans l'une de ces quatre places, ceci nous permet de garantir que l'état du système sera toujours modélisé correctement.

L'invariant de transition nous permet de vérifier que si l'un des transitions de ce groupement est franchissable pendant l'état initial, alors il y aura toujours une transition franchissable parmi les transitions de ce groupement. Si nous considérons l'exemple de la Figure 7.12 comme étant l'état initial, l'invariant de transition {`t1`, `t2`, `t3`, `t4`} nous permet de garantir qu'il y aura toujours une de ces quatre transitions qui sera franchissable. Plus concrètement, ceci nous permet de garantir qu'il y aura toujours un bouton actif parmi les quatre boutons accessibles à l'utilisateur (voir section 5.4) et par extension, qu'une action sera toujours disponible à l'utilisateur.

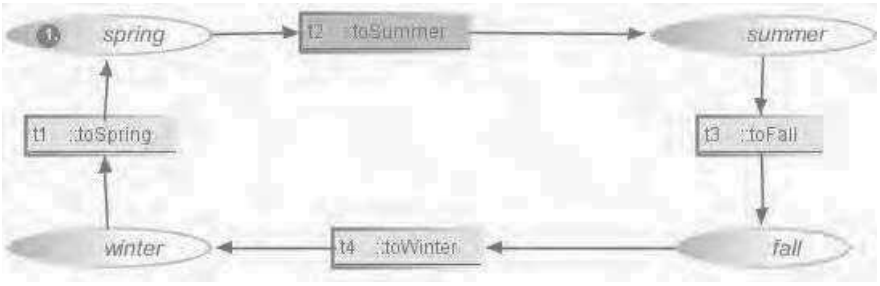


Figure 7.12. Modèle ICO du contrôleur de dialogue de l'application « Les quatre saisons » et le

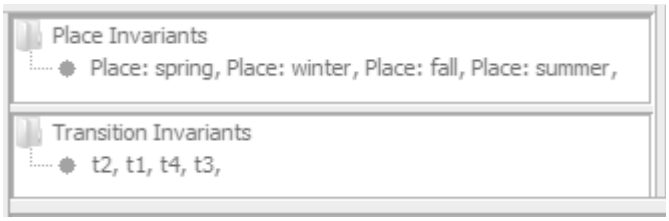


Figure 7.13. Analyse des invariants sur le modèle ICO du contrôleur de dialogue de l'application « Les quatre saisons »

Autres analyses sur les réseaux de Petri sous-jacents aux modèles ICO

Cette section ayant pour but d'illustrer en quoi les analyses formelles peuvent servir les activités de vérification et validation, nous ne présentons pas en détail toutes les analyses pouvant être pratiquées sur les modèles ICO. Ainsi, nous invitons le lecteur intéressé par ces considérations à se référer aux travaux de (Palanque et Rémi 1995) et de (Silva, et al. 2013). Ces travaux présentent comment les analyses structurelles sur les modèles CIO peuvent permettre par exemple d'identifier des interblocages (deadlocks), d'analyser la vivacité des modèles ou de garantir que l'état initial d'un modèle sera toujours accessible.

7.3.1.3 Model-checking sur les modèles ICO

Les analyses que nous venons de présenter sont effectuées sur le réseau de Petri sous-jacent au modèle ICO. Cependant, la notation ICO est beaucoup plus riche qu'un réseau de Petri (voir section 5.2). Ceci signifie que les analyses précédentes sont pratiquées sur des modèles simplifiés, plus abstraits que les modèles ICO. Ce besoin d'abstraction implique une perte de données. Par exemple, dans le cas de ces analyses, les jetons sont des entités simples et ne peuvent pas contenir des objets comme c'est le cas dans la notation ICO.

Pour palier à cette perte d'information, les travaux de (Brat, Martinie et Palanque 2013) proposent de faire du model-checking sur les modèles ICO au travers l'utilisation de l'outil JavaPathFinder[14]. L'utilisation de cet outil supporte la vérification et la validation des modèles ICO en étendant les analyses possibles sur les modèles et notamment la vérification d'assertions, la vérification de propriétés définies en logique temporelles ou de détecter des deaddlocks.

7.3.2 Validation des mécanismes de tolérance aux fautes

Afin de déterminer le niveau de sûreté de fonctionnement du système tolérant aux fautes, il est fondamental de valider les mécanismes de tolérance aux fautes mis en œuvre. Plus concrètement, il est essentiel de s'assurer de deux points :

- Qu'ils ne comportent pas de fautes de conception et le développement.
- Que leur mise en œuvre couvre un nombre d'erreur suffisant relativement au modèle de fautes considéré.

[14] JavaPathFinder : http://babelfish.arc.nasa.gov/trac/jpf/ (au 12 mai 2015)

Dans notre cas, le premier cas est couvert par la simplicité des assertions à contrôler et donc par conséquent par la simplicité des contrôleurs d'assertions. Celle-ci implique une très faible probabilité de fautes logicielles de développement et les contrôleurs d'assertions peuvent donc être considérés comme des logiciels zéro-défaut.

La couverture des mécanismes de tolérance aux fautes équivaut dans notre cas à la couverture de détection des contrôleurs d'assertions. Celle-ci est généralement déterminée de *manière expérimentale* à travers une campagne d'*injection de fautes* (Arlat, et al. 1990). Une telle campagne réclame des moyens importants qui n'ont pu être mis en œuvre dans le cadre des travaux que nous présentons. Nous présentons cependant les principes génériques d'une telle approche et dans quelle mesure celle-ci pourrait être appliquée à nos travaux.

7.3.2.1 Généralités sur les différentes techniques d'injection de fautes

L'injection de fautes (Barbosa, et al. 2012) est une technique clef pour l'évaluation de la sûreté de fonctionnement des systèmes. Celle-ci est généralement pratiquée durant les phases de vérification du système mais peut être envisagée dès la phase de conception comme montré par les travaux de (Pintard, et al. 2014).

L'injection de fautes a longtemps été étudiée et a été appliquée avec succès sur différents systèmes. Il existe différents types d'injection de fautes qui peuvent être classifiés en fonction de leur cible. Ces différents types d'injection de fautes, illustrés en Figure 7.14, peuvent être classés en deux grandes catégories :

- L'*injection de fautes physiques* peut être effectuée en injectant des fautes matérielles dans le système (Hardware Implementend Fault Injection - HIFI) ; en simulant ces fautes de manière logicielle (Software Implemented Fault Injection - SWIFI) ou en utilisant des radiations.
- L'*injection de faute dans les modèles*.

Les techniques d'injection de fautes matérielles par radiation peuvent être effectuées par exemple en bombardant le système avec des ions lourds ou des interférences électromagnétiques. Ces techniques sont peu utilisées à l'heure actuelle car elles manquent de contrôlabilité et ne peuvent être répétées de manière significative.

Les techniques d'injection de fautes HIFI correspondent à de l'injection de fautes au niveau des composants matériels du système tels que les ports (Arlat, et al. 1990).

Les techniques d'injection de fautes SWIFI sont aujourd'hui très répandues du fait de leur facilité de déploiement. Elles peuvent être utilisées avant que le système ne soit déployé (Han, Shin et Rosenberg 1995) ou lorsque celui-ci est déployé (Barbosa, Silva et Cunha 2013).

Les techniques d'injection de fautes dans les modèles (Svenningsson, Model-Implemented Fault Injection for Robustness Assessment 2011) sont utilisées lorsqu'il n'y a pas de solution pour injecter des fautes matérielles, par exemple, lorsque le logiciel n'est pas encore déployé sur un matériel spécifique. Ces techniques ont été développées sur Simulink (Svenningsson, Eriksson, et al. 2010) et sur SCADE (Vinter, et al. 2007).

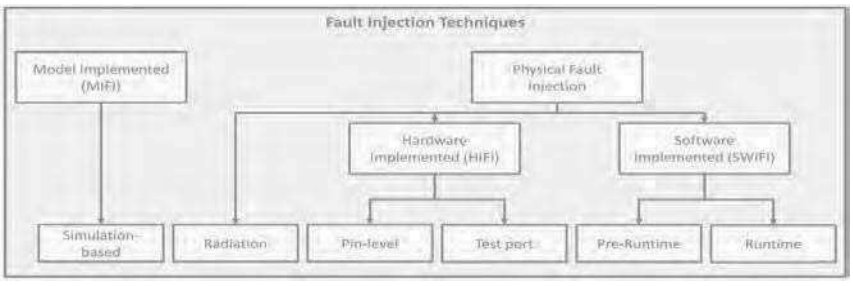


Figure 7.14. Classification des techniques d'injection de fautes (L. Pintard 2015)

7.3.2.2 Injection de fautes dans les modèles ICO

Considérant la mise en œuvre que nous avons proposée pour notre approche (voir section 7.2.4), la validation des mécanismes de tolérance aux fautes peut-être faite en injectant des fautes dans le composant COM du système interactif autotestable. Ce composant étant mis en œuvre par des modèles ICO interprétés en temps réel par l'outil PetShop, il est alors tout à fait envisageable de faire une campagne d'injection de fautes dans les modèles. Les travaux de (Svenningsson, Eriksson, et al. 2010) proposent de simuler les fautes matérielles en modifiant les modèles Simulink de manière à rajouter artificiellement un comportement permettant d'injecter des fautes dans le système.

Ce genre d'injection de fautes est tout à fait envisageable dans les modèles ICO et a été pratiqué dans une moindre mesure par les travaux de (A. Tankeu-Choitat 2011). Cependant, l'injection de fautes dans les modèles ICO peut également être envisagée d'une manière différente en utilisant l'interprète des modèles. En effet, celui-ci a accès à toutes les spécificités des modèles (tels que la valeur des jetons) et peut les modifier durant l'exécution du système. Un exemple d'une telle injection de fautes serait la modification du marquage d'une place : suppression d'un jeton ou modification de l'objet contenu dans le jeton.

Pour pouvoir mener à bien une véritable campagne d'injection de fautes sur les systèmes modélisés par ICO, il s'agit de définir tous les éléments d'un modèle pouvant être affectés par une faute matérielle et de déterminer dans quelle mesure leur modification, suppression ou ajout serait représentatif d'une faute matérielle. Ce travail sort du périmètre des travaux que nous présentons mais sera à terme indispensable pour valider la mise en œuvre de notre approche.

7.3.3 Étude des conflits avec l'utilisabilité

7.3.3.1 Principe de l'approche proposée

Nous avons présenté une approche à base de modèles pour une conception zéro-défaut des systèmes interactifs ainsi qu'une architecture logicielle pour des systèmes interactifs tolérants aux fautes. Il est fondamental de s'assurer que la mise en œuvre de cette approche n'affecte pas le niveau d'utilisabilité du système afin de garantir que le niveau d'utilisabilité du système reste suffisamment élevé.

Selon la norme ICO 9241, l'utilisabilité se décompose en trois facteurs (voir section 2.2.1.2) :

- *L'efficacité* est la capacité à atteindre le but prévu ;
- *L'efficience* est la capacité à atteindre le résultat prévu avec un effort et un temps minimal ;
- *La satisfaction* correspond au confort et au ressenti de l'utilisateur lors de son interaction avec le système.

Dans le domaine des systèmes interactifs critiques, la *satisfaction* n'est pas considérée comme un facteur essentiel car il relève du confort des opérateurs qui est relégué au second plan par rapport à la sécurité-innocuité. Nous ne nous concentrons donc que sur les facteurs *efficacité* et *efficience*.

Comme l'ont montré les travaux de (Hix et Hartson 1993), l'analyse des tâches utilisateur est adaptée pour évaluer l'utilisabilité d'un système. Partant de cette constatation, les travaux de (A. Tankeu-Choitat 2011) proposent de s'appuyer sur l'analyse des tâches utilisateur au travers l'analyse des modèles de tâches afin d'expliciter l'efficacité et l'efficience des systèmes interactifs. Plus précisément, ces travaux ont identifié la notation HAMSTERS (pour Human-centered Assessment and Modeling to Support Task Engineering for Resilient Systems) comme étant particulièrement adaptée dans ce contexte. Cette notation présente en effet des avantages que ne possèdent pas les autres notations permettant de modéliser les tâches utilisateur. Nous retiendrons notamment son pouvoir d'expression, sa capacité de passage à l'échelle (Martinie, Palanque et Winckler 2011) ainsi que sa capacité d'intégration avec les systèmes interactifs modélisés à l'aide de la notation ICO (Barboni, Ladry, et al. 2010). Nous étendons ici ces travaux en proposant une analyse plus poussée des modèles de tâches rendue possible par les évolutions qu'a subi la notation HAMSTERS que nous utilisons.

L'analyse de l'efficacité peut être effectuée en mettant en correspondance les modèles de tâches et le système (Barboni, Ladry, et al. 2010). Cette mise en correspondance permet en effet de vérifier que

tous les services devant être proposés par le système (représentés par les modèles de tâches) sont couverts par les fonctionnalités du système.

L'analyse de l'efficience peut être effectuée en termes de calcul quantitatif de la difficulté d'une tâche. En effet, pour une même tâche, si le nombre de tâches élémentaires devant être accomplies par l'utilisateur est plus important, la charge de travail résultante sera plus importante pour l'utilisateur et donc l'efficience sera affectée. Les nouvelles capacités d'expression de la notation HAMSTERS nous permettent de raffiner ce calcul en permettant de prendre en compte, en plus des tâches effectuées par l'utilisateur, les informations que celui-ci a à retenir. Ces informations ont en effet une importance capitale car elles permettent d'avoir un aperçu de la charge cognitive de l'utilisateur.

Il est important de noter qu'une analyse plus fine sur les tâches peut être réalisée en prenant en compte des modèles plus précis tels que le modèle de processeur humain (Card, et al., 1986) ; nous ne présentons pas cette option ici car elle ne fait pas partie de nos travaux.

L'analyse des tâches se fait principalement sur deux types d'interactions génériques entre l'utilisateur et le système :

- L'*interaction en sortie* qui consiste à la lecture de données par l'utilisateur.
- L'*interaction en entrée* qui consiste à une action de l'utilisateur sur les périphériques d'entrée.

Afin d'expliciter l'approche que nous proposons, nous décrivons dans la sous-section suivante les principes de la notation HAMSTERS puis nous illustrons cette approche sur un exemple simple.

7.3.3.2 Description de la notation HAMSTERS

HAMSTERS[15] est une notation graphique utilisée pour la description de modèles de tâches d'applications coopératives de manière structurée qui a été créée au sein de l'équipe ICS de l'IRIT[16]. Les modèles de tâches peuvent être simulés de manière à étudier les différents chemins d'interactions possibles.

La notation s'appuie sur quatre types de tâches qui peuvent être affinés en sous-types de tâches comme présenté dans le Tableau 7.1. Nous retrouvons :

- *Les tâches abstraites* sont des tâches composées de sous-tâches pouvant être de types différents.
- *Les tâches systèmes* représentent les tâches effectuées en interne par le système.
- *Les tâches utilisateurs* correspondent à des actions effectuées par l'utilisateur indépendamment du système. Il peut s'agir de tâches motrices, cognitives ou perceptives.
- *Les tâches interactives* correspondent à des actions de l'utilisateur sur le système (input), à des actions du système vers l'utilisateur (output) ou aux deux en parallèle (input/output).

Task type	Icons in HAMSTERS task model
Abstract task	 Abstract task
System task	 System task
User task	 User task  Perceptive task  Cognitive task  Motor task
Interactive task	 Interactive input task  Interactive output task  Interactive input output task

Tableau 7.1. Éléments graphiques de la notation HAMSTERS

[15] L'outil HAMSTERS ainsi qu'un manuel d'utilisation et une description de la notation sont disponibles sur : http://www.irit.fr/recherches/ICS/softwares/hamsters/ (au 12 mai 2015)

[16] Équipe ICS de l'IRIT : http://www.irit.fr/recherches/ICS/ (au 12 mai 2015)

Les tâches sont organisées en une hiérarchie structurelle et peuvent être reliées au travers d'opérateurs temporels définissant qualitativement les entrelacements, la séquence, ou le choix entre deux tâches. Ces opérateurs sont présentés dans le Tableau 7.2.

Operator type	Symbol	Description
Enable	T1>>T2	T2 is executed after T1
Concurrent	T1|||T2	T1 and T2 are executed at the same time
Choice	T1[]T2	T1 is executed OR T2 is executed
Disable	T1[>T2	Execution of T2 interrupts the execution of T1
Suspend-resume	T1|>T2	Execution of T2 interrupts the execution of T1, T1 execution is resumed after T2
Order Independent	T1|=|T2	T1 is executed then T2 OR T2 is executed then T1

Tableau 7.2. Opérateurs temporels de la notation HAMSTERS

La notation HAMSTERS permet également de représenter comment certains objets (tels que des données ou des informations) sont liés à certaines tâches. La Figure 7.15 met en évidence comment certains objets peuvent être nécessités en entrée (Figure 7.15-a) d'une tâche ; comment is peuvent être générées ou modifiées par une tâche (Figure 7.15-b) ou les deux simultanément (Figure 7.15-c). Les flots d'objets peuvent également être représentés par des ports d'entrée et de sortie comme présenté en Figure 7.15-d.

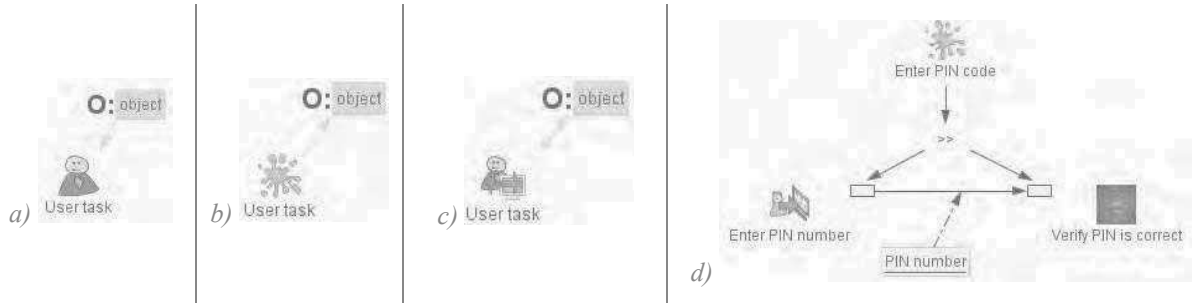


Figure 7.15. Relations entre les tâches et les objets dans HAMSTERS

Afin de représenter la complexité des tâches de l'utilisateur, la notation HAMSTERS permet également de représenter des informations additionnelles telles que les connaissances que l'utilisateur doit avoir pour effectuer la tâche ou les périphériques utilisés pour effectuer la tâche. Ces informations sont représentées en Figure 7.16 où nous pouvons retrouver les connaissances déclaratives (Figure 7.16-a) qui peuvent être affinées en connaissances stratégiques (Figure 7.16-b) ou de situation (Figure 7.16-c) ; les informations (Figure 7.16-d) et les périphériques (Figure 7.16-e).

Figure 7.16. Informations additionnelles dans les modèles de tâche HAMSTERS

7.3.3.3 Illustration sur un exemple simple

La Figure 7.17 présente le modèle de tâche correspondant à l'édition d'une valeur sur un système interactif de cockpit d'avion civil. Les actions pour modifier la valeur d'un paramètre (tâche `Input value`) sont à effectuer de manière séquentielle (opérateur >>).

La tâche commence par l'affichage de la valeur du paramètre par le système sur l'écran (tâche `Display value` et périphérique `Screen`). Cet affichage est perçu par le pilote (tâche `Perceive value`) qui analyse l'information perçue (information `Current value`). Après avoir analysé cette information, le pilote décide de changer la valeur de ce paramètre (tâche `Decide to change value`) en une nouvelle valeur représentée par l'information `New value`. Afin de pouvoir éditer la valeur de ce paramètre, le pilote doit effectuer les tâches suivantes :

- Localiser le périphérique d'entrée (tâche `Locate KCCU` et périphérique `KCCU`). Cette tâche peut être décomposée en plusieurs tâches permettant de représenter le fait que pilote doit tout d'abord localiser le périphérique d'entrée (de manière visuelle ou tactile) et qu'il doit ensuite l'atteindre pour pouvoir l'utiliser.
- Déplacer le curseur graphique sur le widget utilisé pour modifier la valeur du paramètre (tâche `Move cursor to widget`). Cette tâche peut également être affinée en tâches de plus bas niveau car le pilote doit utiliser le périphérique d'entrée pour déplacer le curseur tout en vérifiant la position du curseur sur l'écran jusqu'à avoir atteint le widget concerné.
- Cliquer sur le widget (tâche `Click on widget`). Cette tâche peut être décomposée en plusieurs tâches motrices et perceptives de bas niveau.

Une fois ces tâches accomplies, le pilote peut éditer la valeur du paramètre (tâches `Type value` et `Input value`). Ces actions entrainent une mise à jour du système (tâches `Update value` et `Display value`) qui amènent à la perception de la nouvelle valeur du paramètre par l'utilisateur qui peut alors analyser cette valeur pour la vérifier (tâches `Perceive value` et `Analyse that value is OK`).

Action	Total number of tasks	Number of input – output devices to use	Number of information and its use	Number of user and interactive tasks	Computed tasks difficulty
Input value	13	1 Input + 1 Output (used 7 times)	2 (used 8 times)	10 (3 cognitive, 2 perceptive, 1 motor, 1 interactive, 3 abstract)	(8)+(3+2+1+1+(3*3)) 24

Tableau 7.3. Évaluation de l'utilisabilité en s'appuyant sur les modèles de tâches pour l'édition d'une valeur

Le Tableau 7.3 présente les indicateurs de l'utilisabilité pour le modèle de tâche présenté en Figure 7.17. Chaque colonne de ce tableau présente différentes analyses du modèle de tâche afin de permettre d'évaluer la complexité relative de la tâche abstraite `Input value` :

- Le nombre total de tâches.
- Le nombre de périphériques d'entrée et de sortie ainsi que le nombre de fois qu'ils sont utilisés.
- Le nombre d'informations et leur nombre d'utilisations.
- Le nombre de tâches utilisateur.
- La difficulté relative de la tâche. Elle correspond à la somme des éléments correspond à la charge de travail de l'utilisateur lorsqu'il effectue cette tâche : nombre d'utilisation des informations et nombre de tâches utilisateurs. Nous avons appliqué un coefficient de trois sur les tâches abstraites car elles peuvent être en moyenne décomposées en trois tâches utilisateur.

Le Tableau 7.3 nous permet de déduire que la difficulté relative de la tâche `Input value` est de 24. Cette valeur ne fournit qu'une évaluation quantitative sur la taille et le contenu du modèle de tâche. Cette valeur ne fournit aucune indication valable lorsqu'elle est utilisée seule. En effet, le but de l'approche que nous proposons n'est pas d'évaluer de manière précise l'utilisabilité mais de permettre de comparer l'utilisabilité de deux systèmes afin de s'assurer que les approches utilisées pour augmenter la sûreté de fonctionnement d'un système n'impactent pas négativement l'utilisabilité de celui-ci. Un exemple de l'utilisation de cette approche pour la comparaison de l'utilisabilité de quatre systèmes interactifs avec des niveaux de sûreté de fonctionnement différents est disponible dans (Fayollas, et al. 2014).

Il est également important de noter que la valeur donnée pourrait être plus précise si nous considérions la décomposition des tâches complexes (telles que la localisation du périphérique d'entrée) en utilisant les connaissances du domaine de la cognition et de la psychologie telles que celles fournies par le modèle du processeur humain (Card, Newell et Moran 1983).

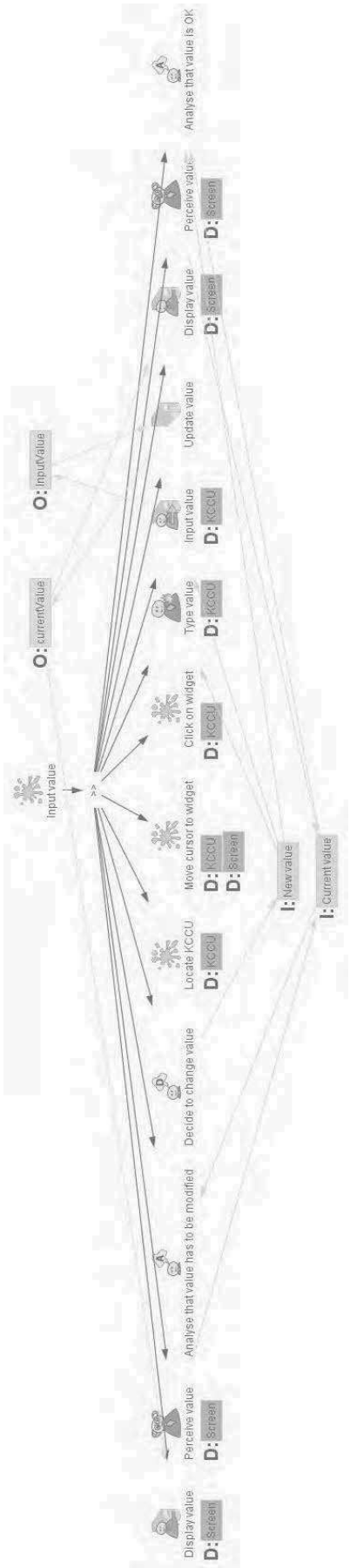


Figure 7.17. Modèle de tâche correspondant à l'édition d'une valeur

7.4 Conclusion

Ce chapitre a tout d'abord présenté l'outil PetShop dont nous proposons l'utilisation pour la mise en œuvre de l'approche zéro-défaut. Cet outil permet l'édition de modèles ICO, leur exécution en temps réel ainsi que leur analyse.

Nous avons ensuite présenté un moyen pour la mise en œuvre de notre architecture logicielle tolérante aux fautes. Cette mise en œuvre s'appuie sur l'utilisation d'ARISSIM, un simulateur de système d'exploitation avionique permettant l'isolation des différents composants logiciels de l'architecture tolérante aux fautes afin de garantir l'isolation de ces composants.

Enfin, nous avons présent des pistes pour effectuer la validation de notre approche. Premièrement, nous avons présenté des pistes pour l'analyse, la validation et la vérification des modèles ICO. Celles-ci sont à étudier et indispensables lorsque l'on applique un processus de développement complet. Deuxièmement, nous avons présenté une piste pour effectuer une campagne d'injection de fautes sur les modèles ICO afin de valider les mécanismes de tolérance aux fautes mis en œuvre. Troisièmement, nous avons présenté une approche s'appuyant sur les modèles de tâches pour comparer l'utilisabilité de deux systèmes et évaluer ainsi l'impact de l'utilisation de notre approche sur l'utilisabilité du système. Cette approche permet de vérifier que les modifications mises en place pour augmenter le niveau de sûreté de fonctionnement n'impactent pas l'utilisabilité du système. Elle permet donc de valider notre approche par rapport à l'utilisabilité du système.

Partie 3. MISE EN ŒUVRE DE L'APPROCHE SUR UNE ÉTUDE DE CAS

Contenu de la partie

Cette troisième partie présente la validation de la faisabilité de notre approche. Celle-ci est effectuée à travers l'application de nos travaux à une étude de cas réaliste issue du milieu de l'avionique : un système interactif inspiré du Flight Control Unit, le système permettant de commander et contrôler le pilote automatique de l'avion et ses différents paramètres.

Le Chapitre 8 est consacré à la présentation de l'étude de cas. Il présente ainsi le système de commande et contrôle Flight Control Unit (FCU) ainsi que l'application interactive (appelée FCUS) que nous avons créée en nous inspirant de celui-ci.

Le Chapitre 9 présente la mise en œuvre de l'approche à base de modèles pour la conception et le développement zéro-défaut de cette étude de cas. Il présente ainsi la modélisation des différents composants de notre application FCUS grâce à la notation ICO.

Le Chapitre 10 présente la mise en œuvre de l'architecture logicielle tolérante aux fautes à cette étude de cas. Il instancie ainsi l'architecture logicielle tolérante aux fautes sur notre application FCUS grâce à la création d'un composant effectuant le contrôle de certaines assertions et à l'implantation de l'application tolérante aux fautes sur un simulateur de système d'exploitation avionique.

Sommaire

Chapitre 8. Présentation de l'étude de cas

Sommaire

Dans cette troisième partie, nous cherchons à valider la faisabilité de notre approche sur une application critique de taille réelle dans un contexte opérationnel valide. Ce chapitre présente l'étude de cas sur laquelle nous nous sommes appuyés afin d'accomplir cet objectif. Cette étude de cas est une application interactive appelée FCUS que nous considérons en remplacement du système de commande et contrôle Flight Control Unit (FCU). Dans ce contexte, cette application est une application critique sur laquelle nous pouvons mettre en œuvre notre approche à base de modèle et notre architecture logicielle.

La première section définit les objectifs recherchés dans l'application de notre approche à une étude de cas. Ces objectifs permettent de définir les critères que doit remplir l'étude de cas choisie et nous amènent à choisir le FCUS.

La deuxième section présente le Flight Control Unit (FCU) de l'A380 ainsi que sa situation dans le cockpit. C'est de ce système que nous nous sommes inspirés pour créer notre application FCUS.

La troisième section présente le FCUS (pour *Flight Control Unit Software*), une application interactive que nous avons construite à partir du FCU.

Enfin, la quatrième section présente en détail le fonctionnement du FCUS.

8.1 Objectifs et mise en garde

Dans cette troisième partie, nous cherchons à valider la faisabilité de notre approche sur une application critique de taille réelle dans un contexte opérationnel valide. Afin d'accomplir cet objectif, l'étude de cas choisie doit répondre à plusieurs critères :

- *Avoir un contexte opérationnel valide* en représentant un système de commande et contrôle critique.
- *Représenter les perspectives d'évolution du cockpit* en représentant le désir de remplacement des systèmes de commande et contrôles physiques par des systèmes interactifs.
- *Représenter l'ensemble des flots de contrôle et d'affichage* qui existent dans les systèmes interactifs respectant le standard ARINC 661.
- *Représenter l'ensemble des types de widgets* utilisés dans les cockpits à l'heure actuelle.

Cependant, nous n'avons pas pour objectif de justifier l'utilisabilité ni le design de l'application que nous avons choisis. En effet, nous rappelons que le design et l'ergonomie des interfaces sont hors du périmètre de notre étude.

Enfin, nous rappelons que nous ne traitons pas ici de la validation de notre approche en termes de vérification et validation de la modélisation du système, de validation de couverture des mécanismes de

tolérance aux fautes ou de validation par rapport aux fautes affectant l'utilisabilité du système. Des pistes pour la validation de notre approche par rapport à ces trois points sont présentées en section 7.3

8.2 Le Flight Control Unit (FCU)

Le Flight Control Unit (ou FCU) est le système de commande et contrôle qui permet de configurer les écrans de pilotage (PFD) et de navigation (ND) ainsi que de paramétrer le pilote automatique. Le FCU est situé à l'avant du cockpit, au dessus des écrans dont il permet le contrôle (PFD et ND) (voir Figure 8.1).

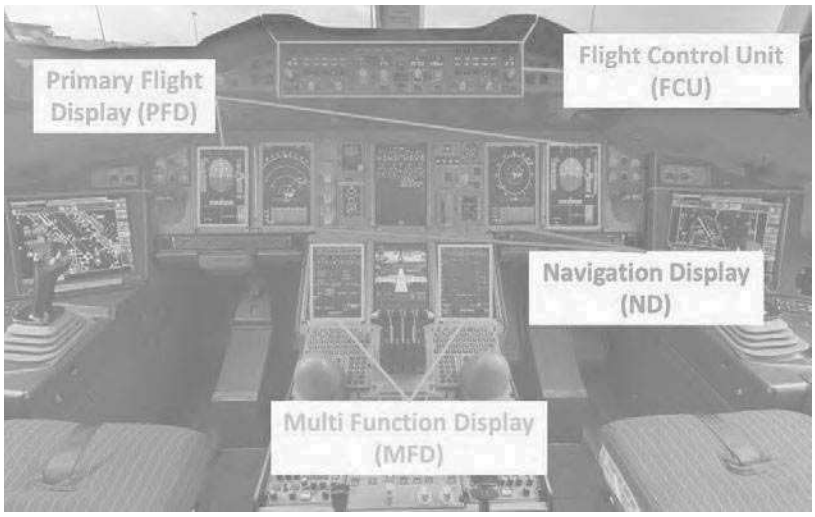


Figure 8.1. Position du Flight Control Unit dans l'environnement du cockpit de l'A380

La Figure 8.2 présente en détail le FCU. Celui-ci est un panneau de commande physique composé de plusieurs boutons poussoir, de commutateurs rotatifs et d'affichages électroniques. Il est constitué de deux types de panneaux de commande :

- *Les EFIS Control Panel* (pour *Electronic Flight Information System*) permettent de configurer les écrans de pilotage et de navigation (PFD et ND) et de régler la pression atmosphérique de référence. Ils sont au nombre de deux, celui de gauche étant réservé au pilote et celui de droite étant réservé au co-pilote.
- L'*AFS Control Panel* (pour *Auto Flight System*) permet la configuration et l'activation du pilote automatique.

Figure 8.2. Flight Control Unit (FCU)

Une grande partie des fonctions commandées et contrôlées par le FCU sont critiques et les systèmes qu'il permet de commander et contrôler (tels que l'autopilote) intègrent des mécanismes de tolérance aux fautes. Ainsi, le FCU est un système interactif critique qui, par conséquent, intègre également des mécanismes de tolérance aux fautes : le câblage des différents boutons est ségrégué, redondant et diversifié et permet de détecter et recouvrir les erreurs pouvant l'affecter.

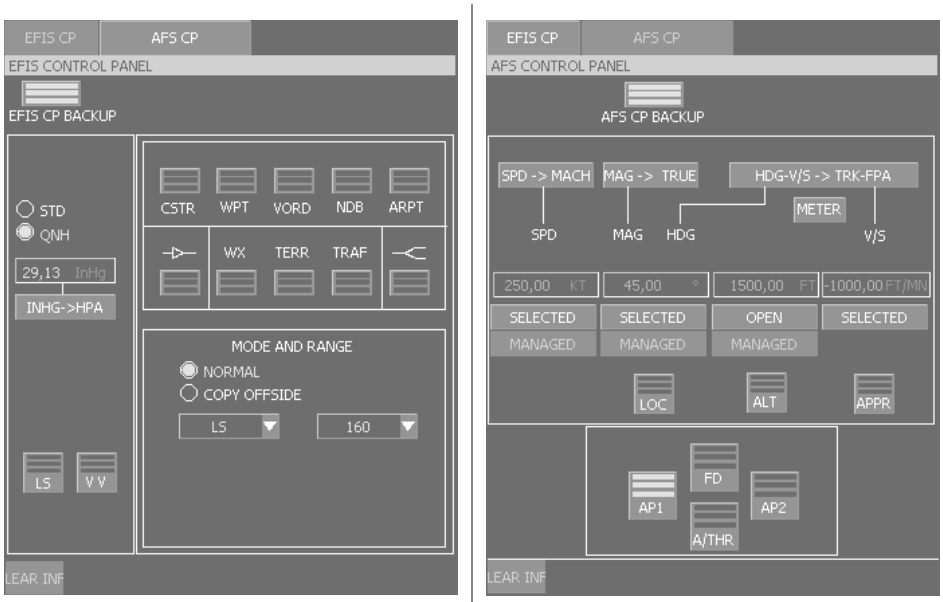
Dans l'A380, en cas de défaillance du FCU, les pilotes peuvent interagir avec une interface logicielle appelée *FCU Backup*. Cette application peut être affichée sur les deux écrans MFD (pour

Multi Function Display) du cockpit (voir Figure 8.1) et respecte le standard ARINC 661. Elle n'est pas critique car elle n'est utilisé qu'en cas de défaillance du FCU, et donc, dans des contions déjà dégradées.

8.3 Le FCUS : une application interactive inspirée du FCU

Le FCU est un système de commande et contrôle critique et l'existence du FCU Backup prouve qu'il est possible de réaliser ses fonctionnalités grâce à une application interactive. Ces deux raisons en font un excellent candidat lorsque l'on considère la possibilité de remplacer les systèmes de commande et contrôles physiques comme le FCU par des systèmes interactifs.

Dans cette optique, nous avons créée une application interactive, appelée FCUS (pour Flight Control Unit Software), permettant aux pilotes de réaliser les mêmes actions que le FCU. Nous nous plaçons dans un contexte opérationnel où cette application est utilisée en remplacement du FCU physique. Dans ce contexte, le FCUS est donc une application critique.



a) EFIS CP *b) AFS CP*

Figure 8.3. Les deux pages interactives de l'application FCUS

L'application FCUS, présentée en Figure 8.3, s'inspire du design du FCU Backup. Elle est constituée de deux pages interactives : la première, appelée EFIS_CP (pour *EFIS Control Panel*) correspond au panneau de contrôle EFIS du FCU physique et la seconde, appelée AFS_CP (pour *AFS Control Panel*), correspond au panneau de contrôle AFS du FCU physique.

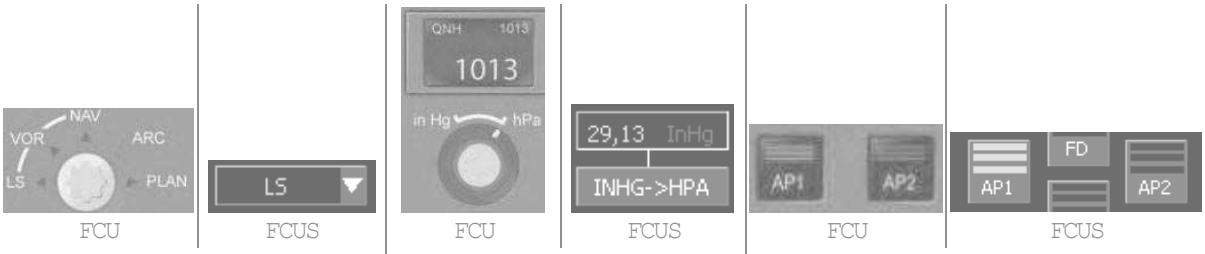
8.4 Fonctionnement de l'application FCUS

Le comportement de notre application est un comportement arbitraire que nous avons choisi en s'inspirant de celui du FCU et des différentes fonctionnalités qu'il offre. De plus, il est important de noter que nous présentons seulement le comportement de l'application FCUS, nous ne présentons pas les transferts d'information avec le noyau fonctionnel qui permettront d'effectuer de manière effective les modifications désirées par le pilote sur les écrans PFD, ND ou sur l'autopilote.

Le FCUS propose ainsi les mêmes commandes et contrôles que le FCU mais avec des interacteurs différents. Ainsi, la Figure 8.4 présente les interacteurs du FCU et du FCUS pour différents types de fonctionnalités disponibles sur ces deux systèmes :

- *Choix d'une option parmi un ensemble de données*. La Figure 8.4-a présente l'exemple de la sélection du mode d'affichage de l'écran ND. Celle-ci est effectuée sur le FCU à l'aide d'un commutateur rotatif et sur le FCUS à l'aide d'une ComboBox.

- *Saisie d'une valeur*. La Figure 8.4-b présente l'exemple de la saisie de la pression atmosphérique de référence. Celle-ci est effectuée sur le FCU à l'aide d'un commutateur rotatif et sur le FCUS à l'aide d'une EditBoxNumeric.
- *Envoi d'une commande*. La Figure 8.4-c présente l'exemple de l'engagement de l'autopilote. Celui-ci est effectué sur le FCU à l'aide d'un bouton poussoir et sur le FCUS à l'aide d'un PicturePushButton.
- *Contrôle de l'état des systèmes* commandés à travers le contrôle de l'état des différents boutons, commutateurs et affichages électroniques.



a) Interacteurs permettant le choix du mode d'affichage du ND

b) Interacteurs permettant de saisir la valeur de la pression atmosphérique de référence

c) Interacteurs permettant d'engager l'autopilote

Figure 8.4. Les interacteurs du FCU et du FCUS

8.4.1 Fonctionnalités de la page EFIS_CP

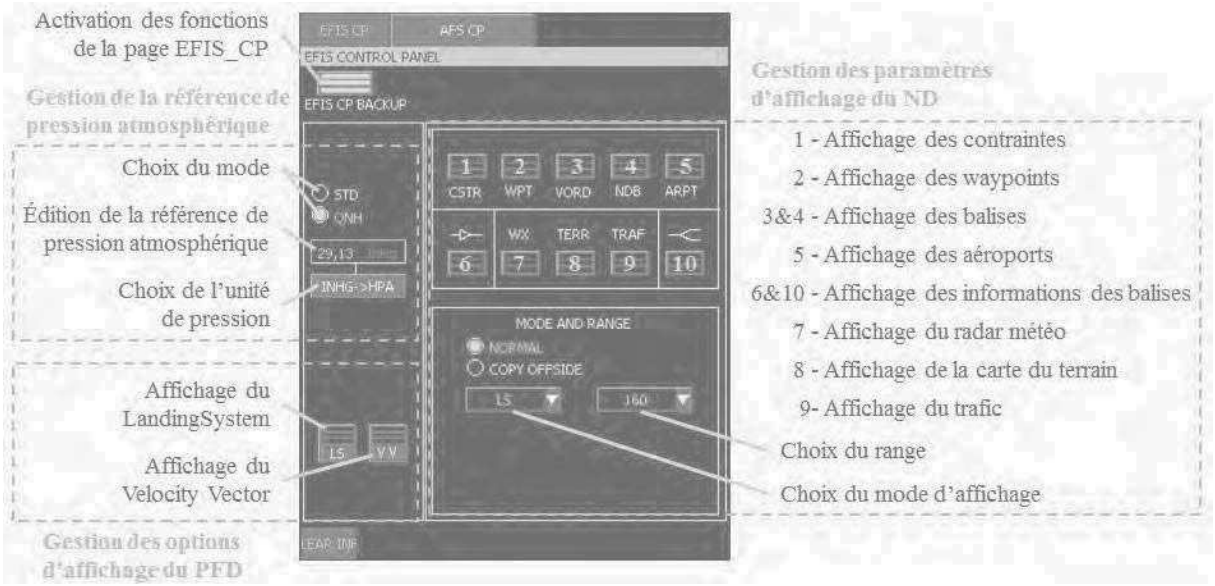


Figure 8.5. Fonctionnalités de la page EFIS_CP

La Figure 8.5 présente les commandes et contrôles disponibles sur la page EFIS_CP du FCUS. Celles-ci peuvent être divisées en trois groupes.

Gestion de la référence de pression atmosphérique.

Deux modes sont disponibles pour la référence de pression atmosphérique : le mode standard (STD) et le mode QNH. Le pilote peut changer de mode à l'aide des *CheckButton* STD et QNH. Le mode standard correspond au mode pendant lequel la référence de pression est égale à la référence standard (équivalente à 29,13 InHg ou 1013,25 hPa). Le mode QNH correspond au mode pendant lequel le pilote peut éditer la référence de pression atmosphérique afin de la caler sur celle de l'aéroport où il va atterrir.

Pour éditer la référence de pression, le pilote saisit, lorsque le mode QNH est activé, la nouvelle référence dans l'*EditBoxNumeric*.

Enfin, le pilote peut modifier l'unité de pression à l'aide du *PicturePushButton* situé sous l'*EditBoxNumeric*.

Gestion des options d'affichage du PFD.

La page EFIS_CP permet l'affichage du LandingSystem (LS) et du Velocity Vector (VV) sur le PFD. L'activation de l'affichage de l'une de ces options est représentée par l'affichage de trois barres vertes sur le PicturePushButton lui correspondant ; sa désactivation par l'affichage de trois barres noires sur celui-ci.

Gestion des options d'affichage du ND.

La page EFIS_CP permet l'affichage de différentes informations sur le ND :

- Affichage des contraintes d'altitude et de vitesse.
- Affichage des waypoints présents aux alentours de l'appareil.
- Affichages des balises très hautes fréquences (VORD) et basses fréquences (NDB) présentes aux alentours de l'appareil.
- Affichage des aéroports présents aux alentours de l'appareil.
- Affichage des informations des balises hautes fréquences.
- Affichage du radar météo.
- Affichage de la carte du terrain.

L'activation de l'affichage de l'une de ces informations est représentée par l'affichage de trois barres vertes sur le *PicturePushButton* lui correspondant ; sa désactivation par l'affichage de trois barres noires sur celui-ci.

Certaines de ces informations ne sont pas affichables sur l'écran en même temps. C'est notamment le cas des deux groupes d'options {WPT, VORD, NDB} et {WX, TERR} parmi lesquels seulement l'une des informations est affichable à la fois.

La page EFIS_CP permet également de choisir le mode d'affichage du ND ainsi que son intervalle. Ceci est effectué à l'aide de deux *ComboBox*.

8.4.2 Fonctionnalités de la page AFS_CP

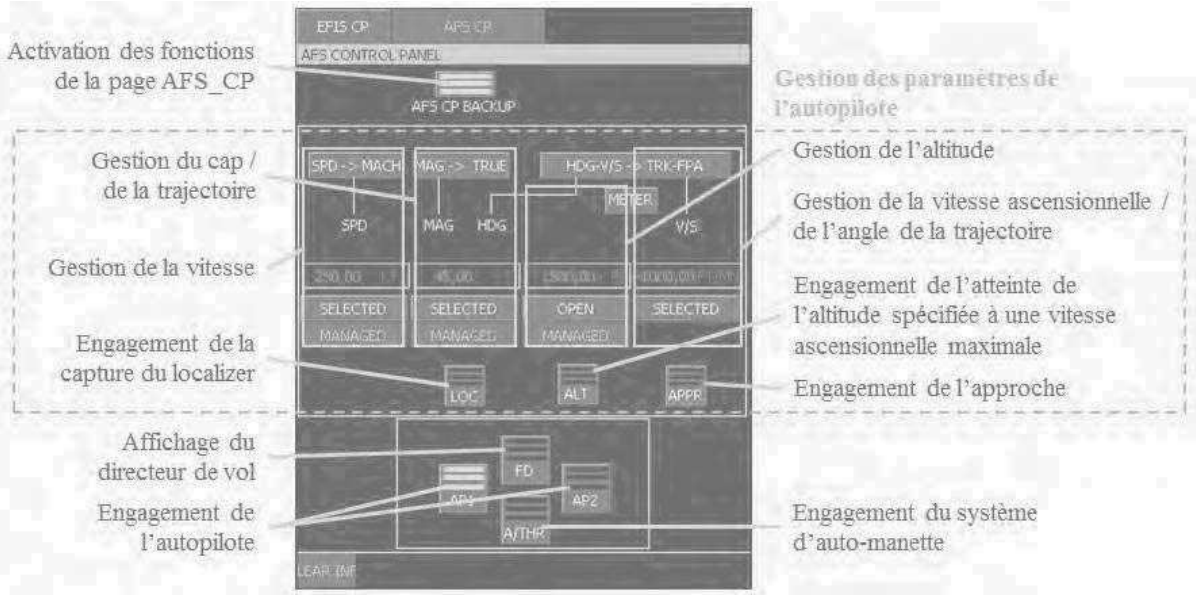


Figure 8.6. Fonctionnalités de la page AFS_CP

La Figure 8.6 présente les commandes et contrôles disponibles sur la page AFS_CP du FCUS. Celles-ci peuvent être divisées en deux groupes.

Gestion de l'autopilote et de ces paramètres

L'engagement de l'autopilote peut être effectué grâce aux *PicturePushButton* AP1 et AP2. Si l'autopilote 1 est engagé, trois barres vertes sont affichées sur le *PicturePushButton* AP1. Si celui-ci est désengagé, trois barres noires sont affichées sur celui-ci. Le comportement est le même pour l'engagement de l'autopilote 2.

Lorsque l'un des deux autopilotes est engagé, le pilote peut les paramétrer en gérant différents paramètres :

- La vitesse.
- Le cap ou la trajectoire.
- L'altitude.
- La vitesse ascensionnelle ou l'angle de trajectoire.

Ces différents paramètres peuvent être gérés selon deux modes : le mode *Managed* et le mode *Selected*. Un paramètre en mode *Managed* est directement géré par le système de commande de vol directement par l'autopilote en fonction du plan de vol qui a été entré avant le départ de l'avion, le pilote ne peut pas modifier sa valeur Dans ce cas, les paramètres sont gérés en mode *Managed* et l'affichage de leur valeur dans les *EditBoxNumeric* est effectué en magenta.

Pour pouvoir modifier la valeur d'un paramètre, le pilote doit passer en mode *Selected*. Pour cela, il rentre une nouvelle valeur dans l'*EditBoxNumeric* correspondante et l'engager en cliquant sur le *PicturePushButton* `Selected` correspondant. Le paramètre est alors géré en mode *Selected* et l'affichage de sa valeur dans l'*EditBoxNumeric* correspondante est effectué en cyan.

Pour chacun de ces paramètres, le pilote peut également modifier leur unité. Ainsi, la vitesse de l'avion peut être affichée en nœuds (unité KT pour knots, mode SPD) ou en mach (mode MACH). Le cap et l'angle de trajectoire peuvent être affichés par rapport au nord magnétique (mode MAG) ou au nord vrai (mode TRUE). Enfin, les références d'altitude (altitude et vitesse ascensionnelle) peuvent être affichées en pieds ou en mètres (FT ou METER).

Le pilote peut également engager les trois paramètres suivants :

- Engagement de la capture du localizer.
- Engagement de l'atteinte de l'altitude spécifiée à la vitesse maximale.
- Engagement de l'approche.

L'activation de l'un de ces paramètres est représenté par l'affichage de trois barres vertes sur le *PicturePushButton* lui correspondant ; sa désactivation par l'affichage de trois barres noires.

Engagement de différentes options

La page AFS_CP permet également les options suivantes :

- L'affichage du directeur de vol sur le PFD.
- L'engagement du système d'auto-manette.

L'activation de l'une de ces options est représentée par l'affichage de trois barres vertes sur le *PicturePushButton* lui correspondant ; sa désactivation par l'affichage de trois barres noires.

8.4.3 Structuration du FCUS

Le FCUS respecte le standard ARINC 661 et est constitué de 123 widgets dont 64 sont critiques. Ils sont de 12 types différents :

- Des objets de regroupement (ou container) : *Layer*, *BasicContainer*, *Panel* et *RadioBox*.

- Des widgets d'action : *CheckButton*, *ComboBox*, *EditBoxNumeric*, *PicturePushButton* et *PictureToggleButton*.
- Des widgets d'affichage : *Label*, *GP_Line* et *Picture*.

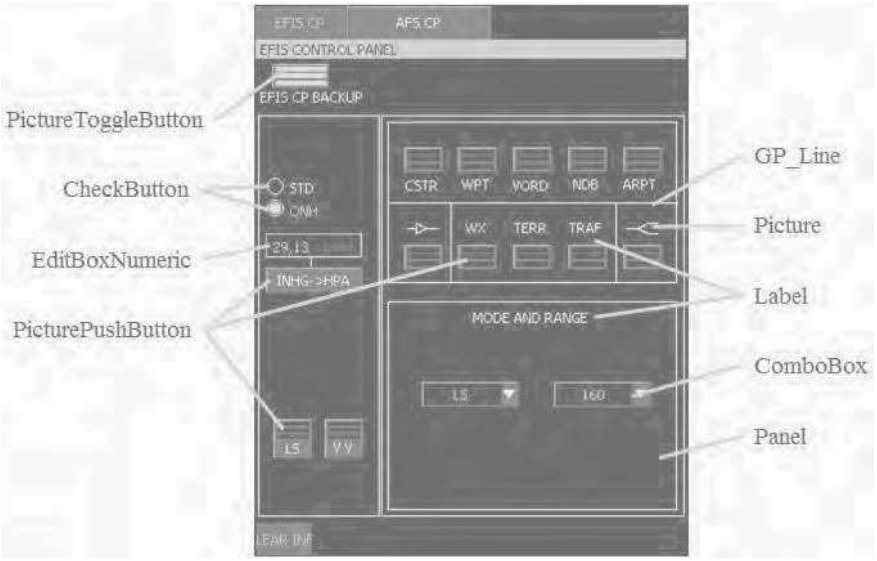


Figure 8.7. Les différents types de widgets de l'application FCUS

Tous ces différents types de widgets sont présents sur la page EFIS_CP. Nous les avons mis en évidence sur la Figure 8.7. Les widgets qui ne sont pas présents sur cette figure ne possèdent pas de représentation graphique (*Layer*, *BasicContainer* et *RadioBox*). Cette figure rend également explicite le fait qu'en fonction de la valeur de leurs paramètres, les widgets peuvent avoir des représentations graphiques différentes. Ceci est particulièrement visible sur cette figure pour les *PicturePushButton* dont nous pouvons voir des exemples avec une image et un label (*PicturePushButton* `LS`), avec seulement un label (*PicturePushButton* `INHG -> HPA`) ou encore avec seulement une image (*PicturePushButton* sous le *Label* `WX`). Le Tableau 8.1 présente la répartition des 123 widgets entre les 12 types et présente pour chaque type de widgets combien d'entre eux sont critiques.

Type de widget	Nombre total de widgets	Nombre de widgets critiques
Layer	1	1
BasicContainer	12	10
Panel	7	4
RadioBox	1	1
CheckButton	4	2
ComboBox	3	0
EditBoxNumeric	8	8
PicturePushButton	40	25
PictureToggleButton	2	2
Label	32	4
GP_Line	11	7
Picture	2	0

Tableau 8.1. Tableau récapitulatif des widgets et de leur type dans le FCUS

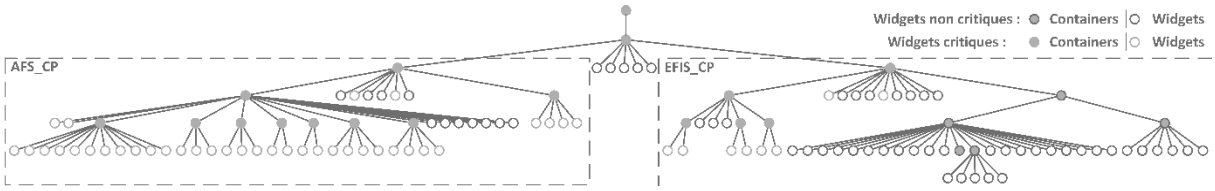


Figure 8.8. Arbre des widgets de l'application FCUA

La structure de l'application FCUS (l'arbre de ces widgets) est présentée en Figure 8.8. Nous ne présentons pas l'arbre des widgets détaillé car celui-ci est très imposant. Cette figure nous donne cependant un aperçu du nombre élevé de widgets qui composent l'application. Les widgets critiques sont présentés avec un bord rouge. Ils correspondent à tous les widgets en rapport avec l'édition de la référence de pression (pour la page EFIS_CP) et l'engagement et la gestion des paramètres de l'autopilote (pour la page AFS_CP) et sont rendus explicites sur la Figure 8.9.

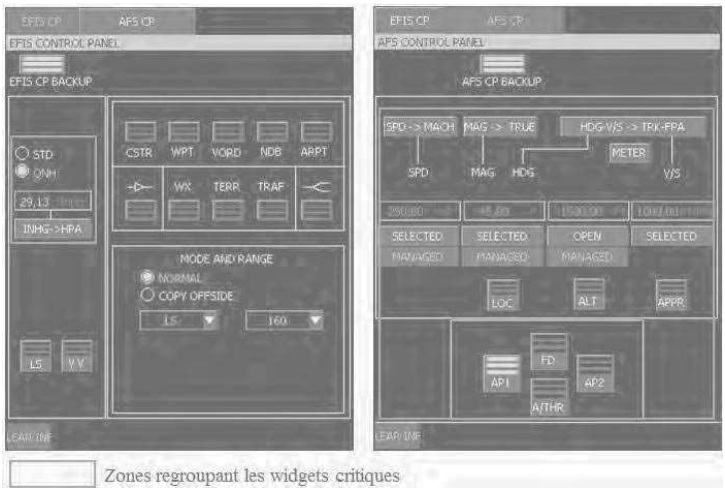


Figure 8.9. Widgets critiques sur les pages EFIS_CP et AFS_CP du FCUS

8.5 Synthèse

Dans ce chapitre, nous avons présenté l'étude de cas qui nous permet de valider la faisabilité de notre approche. Cette étude de cas consiste en une application interactive appelée FCUS. Elle a été construite à partir d'un système de commande et contrôle analogique et de sa version logicielle de secours : le Flight Control Unit (FCU) qui permet de paramétrer les écrans de pilotage et de navigation ainsi que l'autopilote.

Nous nous plaçons dans un contexte opérationnel dans lequel l'application FCUS remplace le FCU. Dans ce contexte, l'application FCUS correspond à un système interactif critique sur lequel nous appliquons notre approche à base de modèle dans le Chapitre 9 et notre architecture logicielle dans le Chapitre 10.

Cette étude de cas remplit les caractéristiques que nous nous sommes fixés pour la validation de la faisabilité notre approche. En effet, elle permet de :

- *Avoir un contexte opérationnel valide* en représentant un système de commande et contrôle critique.
- *Représenter les perspectives d'évolution du cockpit* en représentant le désir de remplacement des systèmes de commande et contrôles physiques par des systèmes interactifs.
- *Représenter l'ensemble des flots de contrôle et d'affichage* qui existent dans les systèmes interactifs respectant le standard ARINC 661.
- *Représenter l'ensemble des types de widgets* utilisés dans les cockpits à l'heure actuelle.

Cependant, cette étude de cas ne nous permet pas la validation de notre approche en termes de vérification et validation de la modélisation du système, de validation de couverture des mécanismes de

tolérance aux fautes ou de validation par rapport aux fautes affectant l'utilisabilité du système. Nous rappelons que des pistes pour la validation de notre approche par rapport à ces trois points ont été présentées en section 7.3.

Chapitre 9. Mise en œuvre de l'approche à base de modèles

Sommaire

Afin d'illustrer la faisabilité de notre approche à base de modèles, ce chapitre présente la mise en œuvre de celle-ci sur notre étude de cas. Nous n'appliquons pas ici un processus précis car ceci est en dehors du périmètre de la thèse comme nous l'avons présenté précédemment. Par contre, nous présentons en détail la modélisation du comportement des différents composants logiciels du système interactif qui ont été identifiés dans l'architecture présentée en section 4.3. Nous avons présenté dans le Chapitre 5 la notation formelle ICO pour la modélisation de ces différents composants. Celle-ci nous permet de réaliser une description complète, concise et non ambigüe du comportement des composants logiciels afin de nous rapprocher d'un logiciel zéro-défaut.

La première section présente l'architecture logicielle de l'étude de cas présentée dans le Chapitre 8.

La deuxième section présente les modèles ICO des différents composants du système interactif intégrant l'application FCUS.

Enfin, la troisième section présente comment cette modélisation a été mise en œuvre grâce à l'utilisation de l'outil PetShop.

9.1 Architecture logicielle de modélisation de l'étude de cas

La Figure 9.1 présente l'instanciation de l'architecture logicielle d'un système interactif que nous avons présentée en section 4.3 à un système interactif respectant le standard ARINC 661 et permettant d'exécuter l'application FCUS.

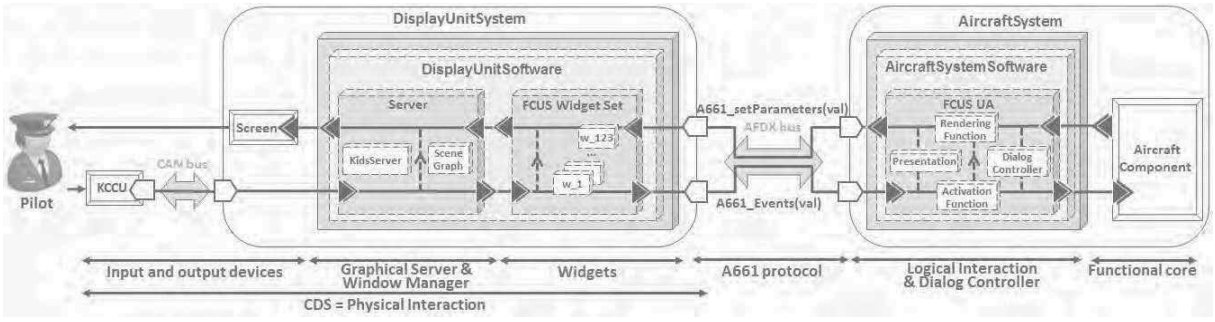


Figure 9.1. Application FCUS dans l'architecture logicielle d'un cockpit interactif

Comme nous l'avons présenté dans le Chapitre 5, notre approche à base de modèles pour concevoir et développer des systèmes interactifs s'approchant autant que possible du zéro-défaut est fondée sur la modélisation des différents composants logiciels à l'aide de la notation formelle ICO. Ainsi, la suite de

ce chapitre est dédiée à la modélisation ICO de tous les composants logiciels représentés par des boîtes blanches dans la Figure 9.1 :

- *Les différents composants logiciels du serveur* : étant donné que notre serveur respecte le standard ARINC 661, il ne propose que très peu de techniques d'interactions et nous avons choisi de le diviser en deux composants. Le premier composant, appelé `KidsServer` est responsable de toutes les fonctionnalités du serveur, à l'exception de la gestion de la hiérarchie des widgets qui est gérée par le second composant, appelé `SceneGraph`.
- *Les différents widgets* : chaque widget du même type possède un comportement similaire et est donc représenté par un modèle ICO.
- *Les différents composants logiciels de l'UA* : elle est divisée en quatre composants logiciels qui sont la présentation, les fonctions d'activation et de rendu et le contrôleur de dialogue.

Nous avons montré dans le Chapitre 5 comment les modèles des fonctions d'activation et de rendu pouvaient être traduits à partir des tableaux qui les définissent. Ainsi, au vu du nombre de modèles à présenter, nous avons choisi pour cette étude de cas de seulement présenter les tableaux correspondant à ces fonctions. Nous ne présentons pas non plus ici le modèle du composant présentation. En effet, pour ces trois modèles (fonction d'activation et de rendu et présentation), les principes de modélisation que nous avons montré pour l'exemple de l'application « les 4 saisons » (voir section 5.4) restent les mêmes dans cette nouvelle étude de cas ; ainsi, nous ne les détaillons pas ici.

9.2 Modélisation des différents composants du FCUS

9.2.1 Modélisation du contrôleur de dialogue

Le contrôleur de dialogue du FCUS est un modèle très imposant qui ne serait pas lisible si nous le présentions de manière complète. Nous avons donc choisi de ne pas le présenter dans son ensemble et de le diviser en plusieurs extraits fonctionnels. Ainsi, nous présentons premièrement la gestion de la page affichée. Deuxièmement, nous présentons les fonctions du FCUS permettant de paramétrer l'écran de pilotage (PFD). Troisièmement, nous présentons les fonctions du FCUS permettant de paramétrer l'écran de navigation (ND). Enfin, nous présentons les fonctions présentes sur la page AFS_CP permettant de paramétrer l'autopilote.

9.2.1.1 Gestion des pages EFIS_CP et AFS_CP

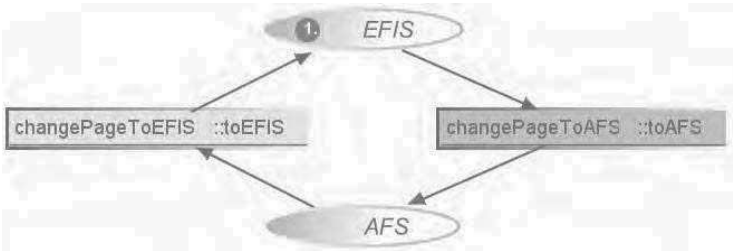


Figure 9.2. Modèle ICO de la gestion des deux pages EFIS_CP et AFS_CP

Le modèle de la gestion de l'affichage des pages EFIS_CP ou AFS_CP est présenté en Figure 9.2. À l'initialisation, la page affichée par défaut est la page EFIS_CP (jeton dans la place `EFIS`). Le changement d'état est déclenché par le franchissement de la transition `changePageToAFS` qui enlève le jeton de la place `EFIS` pour en placer un dans la place `AFS`.

Comme le montre la fonction d'activation traduite dans le Tableau 9.1, un tel changement d'état est provoqué par le clic d'un pilote sur le *PicturePushButton* `PPB_AFS`. Enfin, comme le montre le Tableau 9.2, la modification de l'affichage de l'application est déclenchée par le déplacement du jeton entre les places `EFIS` et `AFS`. Ainsi, si un jeton est présent dans la place `EFIS`, c'est la page EFIS_CP qui est affichée et réciproquement pour la page AFS_CP.

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_EFIS	Clic	toEFIS	changePageToEFIS
PPB_AFS	Clic	toAFS	changePageToAFS

Tableau 9.1 Fonction d'activation pour la gestion des pages EFIS_CP et AFS_CP

État	Place	Événement(s)	Rendu associé
EFIS	EFIS	Jeton entré Marquage réinitialisé	Affichage de la page EFIS_CP
AFS	AFS	Jeton entré	Affichage de la page AFS_CP

Tableau 9.2. Fonction de rendu pour la gestion des pages EFIS_CP et AFS_CP

Nous présentons également dans le Tableau 9.3 le rendu d'activation de cette fonction. Celui-ci est réalisé par la fonction d'activation et établit le lien entre l'activation des objets d'interaction et les event handler associés : si un event handler est activé (respectivement désactivé), l'objet d'interaction qui lui est associé doit être actif (respectivement inactif). Ce tableau étant redondant avec celui de la fonction d'activation, nous ne le présenterons pas pour les fonctions suivantes.

Objet d'interaction	Event handler	Événements	Rendu associé
PPB_EFIS	toEFIS	Activation/Désactivation de l'event handler	Activation/Désactivation de l'obet d'interaction
PPB_AFS	toAFS	Activation/Désactivation de l'event handler	Activation/Désactivation de l'obet d'interaction

Tableau 9.3. Rendu d'activation pour la gestion des pages EFIS_CP et AFS_CP

9.2.1.2 Gestion du panneau pour le paramétrage du PFD (page EFIS_CP)

Le panneau de la page EFIS_CP consacré au paramétrage du PFD présente deux fonctions principales que nous présentons ci-dessous : la gestion de la référence de pression atmosphérique et la gestion des options d'affichage du PFD.

Gestion de la référence de pression atmosphérique

Deux modes sont disponibles pour la référence de pression atmosphérique : le mode standard (STD) et le mode QNH. Le mode standard correspond au mode pendant lequel la référence de pression est égale à la référence standard (équivalente à 29,13 InHg ou 1013,25 hPa). Le mode QNH correspond au mode pendant lequel le pilote peut éditer la référence de pression atmosphérique afin de la caler sur celle de l'aéroport où il va atterrir.

Le choix du mode est géré par deux *CheckButton* qui sont maintenus en exclusivité mutuelle par une *RadioBox* (voir section 9.2.2.2). L'unité de pression peut être modifiée en hPa (respectivement InHg) grâce à l'utilisation du *PicturePushButton* `INHG -> HPA` (respectivement `HPA -> INHG`). Enfin, en mode QNH, le pilote peut modifier la valeur de la référence de pression atmosphérique grâce à l'utilisation de l'*EditBoxNumeric*.

La Figure 9.3 présente le modèle ICO de la gestion de la référence de pression atmosphérique, le Tableau 9.4 présente la fonction d'activation associée et le Tableau 9.5 présente la fonction de rendu associée. L'édition de la valeur de la référence de pression atmosphérique est possible lorsque l'interface est en mode QNH (jeton dans la place `QNH`). Dans ce cas, si l'unité de pression est InHg (jeton dans la place `InHg`), la modification de la valeur de la pression dans *l'EditBoxNumeric* permet le franchissement de la transition `changeInHgtempValue`, ce qui placera la nouvelle valeur de la pression (`temp`) dans la place `NewInHg`. Cette valeur est alors testée. Si cette valeur est inférieure à la valeur minimale autorisée ou supérieure à la valeur maximale autorisée, elle est corrigée à la valeur minimale ou maximale (transitions `correctInHg1` et `correctInHg2`) ; sinon, aucune correction n'est effectuée (transition `correctInHg3`). Le comportement est similaire si l'unité de pression est hPa (jeton dans la place `hPa`).

Si le pilote décide de changer l'unité de pression (ici de InHg vers hPa), il clique sur le bouton correspondant. Cette action se traduit par le franchissement de la transition `switchToHPA` qui recalcule la nouvelle valeur de la référence de pression en hPa.

Les différents rendus graphiques associés à ce comportement sont présentés en Figure 9.4.

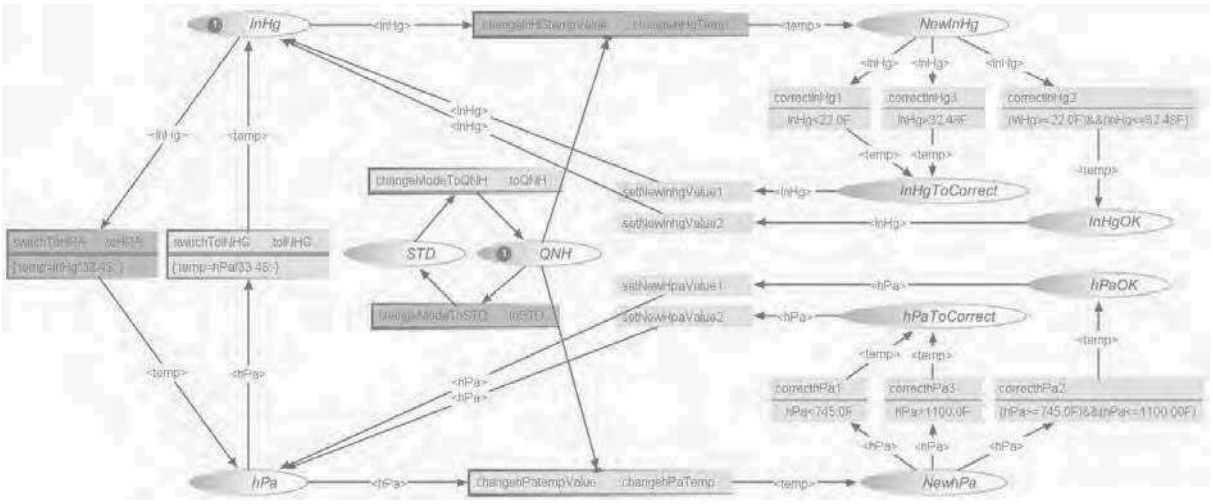


Figure 9.3. Modèle ICO de la gestion de la référence de pression atmosphérique

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_INHGtoHPA	Clic	toHPA	switchToHPA
PPB_HPAtoINHG	Clic	toINHG	switchToINHG
EBN_INHG	Clic & édition de la valeur	ChangeInHgTemp	changeInHgTempValue
EBN_HPA	Clic & édition de la valeur	ChangehPaTemp	changehPaTempValue

Tableau 9.4 Fonction d'activation pour la gestion de la référence de pression

État	Place	Événement(s)	Rendu associé
InHg	INHG	Jeton entré Marquage réinitialisé	Affichage de la référence de pression en InHg
hPa	HPA	Jeton entré	Affichage de la référence de pression en hPa

Tableau 9.5. Fonction de rendu pour la gestion de la référence de pression

a) Mode QNH et unité InHg | *b) Mode STD et unité InHg* | *c) Mode QNH et unité hPa* | *d) Mode STD et unité hPa*

Figure 9.4. Les différents rendus graphiques associés à la gestion de la référence de pression atmosphérique

Gestion des options d'affichage du PFD

La page EFIS_CP propose deux options d'affichage pour le PFD : l'affichage du *LandingSystem* et l'affichage du *VelocityVector*. Les affichages de ces deux options sont commandés par deux *PicturePushButton* affichant trois barres noires lorsqu'ils sont désactivés ou trois barres vertes lorsqu'ils sont activés. Le modèle ICO de la gestion de ces deux PicturePushButton est présenté en Figure 9.5, la fonction d'activation correspondante est présentée dans le Tableau 9.6 et la fonction de rendu dans le Tableau 9.7.

Si l'on prend l'exemple de l'affichage du *LandingSystem*, les places `LS_OFF` et `LS_ON` représentent respectivement les cas où cet affichage est activé (et donc le *PicturePushButton* correspondant présente trois barres vertes) et celui où il est désactivé (dans ce cas, le *PicturePushButton* correspondant présente trois barres noires). Ces deux différents rendus graphiques sont présentés en Figure 9.6

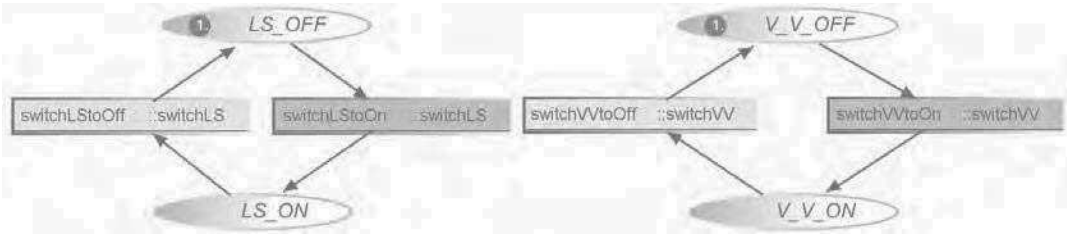


Figure 9.5. Modèle ICO de la gestion des options d'affichage du PFD

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_LS	Clic	switchLS	switchLStoOff, switchLStoOn
PPB_VV	Clic	switchVV	switchVVtoOff, switcVVtoOn

Tableau 9.6 Fonction d'activation pour la gestion des options d'affichage du PFD

État	Place	Événement(s)	Rendu associé
LS désactivé	LS_OFF	Jeton entré Marquage réinitialisé	Affichage des trois barres noires sur PPB_LS
LS activé	LS_ON	Jeton entré	Affichage des trois barres vertes sur PPB_LS
VV désactivé	V_V_OFF	Jeton entré Marquage réinitialisé	Affichage des trois barres noires sur PPB_VV
VV activé	V_V_ON	Jeton entré	Affichage des trois barres vertes sur PPB_VV

Tableau 9.7. Fonction de rendu pour la gestion des options d'affichage du PFD

a) LandingSystem non affiché sur le PFD

b) LandingSystem affiché sur le PFD

Figure 9.6. Rendus graphiques associés à la gestion du PicturePushButton permettant l'affichage du LandingSystem

9.2.1.3 Gestion du panneau pour le paramétrage du ND (page EFIS_CP)

Le panneau de la page EFIS_CP consacré au paramétrage du ND permet l'affichage des paramètres suivants sur l'écran de navigation (voir section 8.4.1) :

- Affichage des contraintes d'altitude et de vitesse.
- Affichage des waypoints présents aux alentours de l'appareil.
- Affichages des balises hautes fréquences (VORD) et basses fréquences (NDB) présentes aux alentours de l'appareil.
- Affichage des aéroports présents aux alentours de l'appareil.
- Affichage des informations des balises hautes fréquences.
- Affichage du weather radar.
- Affichage de la carte du terrain.
- Choix du mode d'affichage du ND.
- Choix du range du ND.

Le comportement correspondant aux affichages de ces informations est similaire, pour la plupart d'entre elles, à celui que nous venons de présenter pour l'affichage du *LandingSystem*. C'est ainsi le cas des informations CSTR, ARPT, des informations des balises hautes fréquences et TRAF. Certaines des options restantes sont en exclusivité mutuelles. Ainsi, il n'est pas possible d'afficher les waypoints ou les balises en même temps, seulement l'une des trois options (WPT, VORD ou NDB) est accessible à la fois. Le modèle de la gestion de ce comportement est présenté en Figure 9.8, la fonction d'activation associée est présentée dans le Tableau 9.8 et la fonction de rendu dans le Tableau 9.9.

Ainsi, lorsqu'aucune des trois options d'affichage n'est activée (jeton avec la valeur `0` dans la place `NDOption`), si le pilote clique sur l'un des trois *PicturePushButton* (`WPT`, `VORD` ou `NDB`), celui-ci présente alors les trois barres vertes pour notifier que l'affichage est activé. Si l'une des options d'affichage est déjà activée (jeton dans la place `NDOption` avec une valeur parmi 1, 2 ou 3), deux options s'offrent au pilote :

- Il peut désactiver l'option en cliquant de nouveau sur le *PicturePushButton* correspondant, ce qui a pour effet de remplacer le jeton dans la place `NDOption` par un jeton avec la valeur `0` et d'afficher trois barres noires sur les trois *PicturePushButton*.
- Il peut choisir d'activer une autre option en cliquant sur le *PicturePushButton* correspondant à celle-ci, ce qui a pour effet de désactiver l'option actuelle et de la remplacer par l'option désirée. Ceci se traduit dans le modèle ICO par le remplacement du jeton dans la place `NDOption` par un jeton avec une autre valeur (`1`, `2` ou `3`) en fonction de la nouvelle option activée).

Les rendus graphiques associés à ces différents états sont présentés en Figure 9.7.

De la même manière, les options WX et TERR sont en exclusivité mutuelles. Nous ne présentons pas ici le modèle ICO correspondant à ce comportement car il est assez similaire à celui que nous venons de présenter et n'apporte rien de nouveau dans les principes de modélisation.

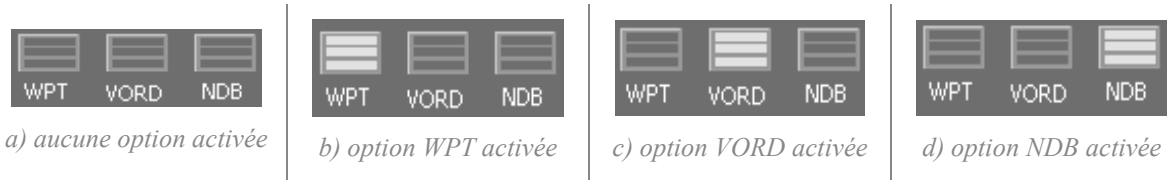


Figure 9.7. Rendus graphiques associés à la gestion des options d'affichage WPT, VORD et NDB

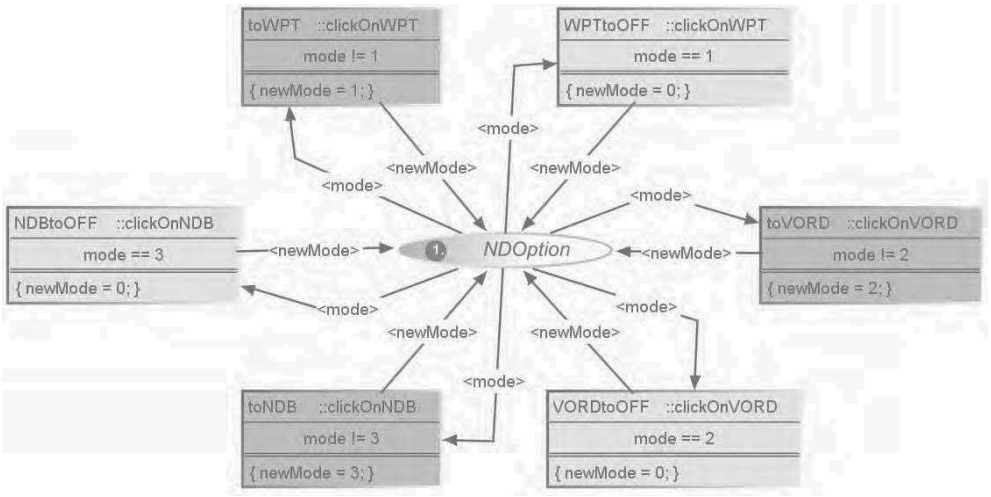


Figure 9.8. Modèle ICO de la gestion de l'exclusivité entre les options d'affichage WPT, VORD et NDB

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_WPT	Clic	clickOnWPT	toWPT, WPTtoOFF
PPB_VORD	Clic	clickOnVORD	toVORD, VORDtoOFF
PPB_NDB	Clic	clickOnNDB	toNDB, NDBtoOFF

Tableau 9.8 Fonction d'activation pour la gestion de l'exclusivité entre les options d'affichage WPT, VORD et NDB

État	Place	Événement(s)	Rendu associé
WPT, VORD et NDB désactivés	NDOption	Jeton entré avec la valeur 0 Marquage réinitialisé	Affichage des trois barres noires sur PPB_WPT, PPB_VORD et PPB_NDB
WPT activé	NDOption	Jeton entré avec la valeur 1	Affichage des trois barres noires sur PPB_VORD et PPB_NDB Affichage des trois barres vertes sur PPB_WPT
VORD activé	NDOption	Jeton entré avec la valeur 2	Affichage des trois barres noires sur PPB_WPT et PPB_NDB Affichage des trois barres vertes sur PPB_VORD
NDB activé	NDOption	Jeton entré avec la valeur 3	Affichage des trois barres noires sur PPB_WPT et PPB_VORD Affichage des trois barres vertes sur PPB_NDB

Tableau 9.9. Fonction de rendu pour la gestion de l'exclusivité entre les options d'affichage WPT, VORD et NDB

Le modèle ICO de la gestion du choix du mode et du range sur l'écran de navigation est présenté en Figure 9.9. La fonction d'activation associée est présentée dans le Tableau 9.10 et la fonction de rendu dans le Tableau 9.11. Si nous prenons l'exemple du mode, lorsque le pilote clique sur la *ComboBox* correspondante et valide une entrée, la transition `changeMode` est franchie. Son franchissement remplace le jeton présent dans la place `Mode` par un jeton contenant une valeur correspondante au nouveau mode. L'entrée d'un jeton dans cette place impose l'affichage du nouveau mode.

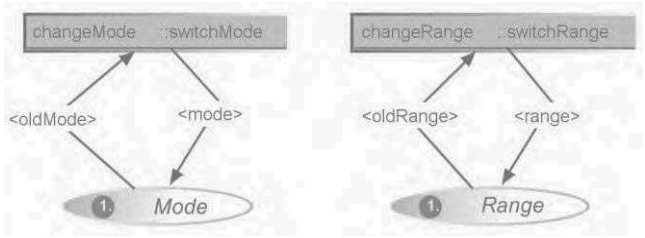


Figure 9.9. Modèle ICO de la gestion du choix du mode et du range

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
ComboBox_mode	Clic & validation d'une entrée	switchMode	changeMode
ComboBox_range	Clic & validation d'une entrée	switchRange	changeRange

Tableau 9.10 Fonction d'activation pour la gestion du choix du mode et du range

État	Place	Événement(s)	Rendu associé
Mode x activé	Mode	Jeton entré avec la valeur x Marquage réinitialisé avec la valeur x	Affichage du mode correspondant à la valeur x
Range x activé	Range	Jeton entré avec la valeur x Marquage réinitialisé avec la valeur x	Affichage du range correspondant à la valeur x

Tableau 9.11. Fonction de rendu pour la gestion du choix du mode et du range

9.2.1.4 Gestion de la page AFS_CP permettant le paramétrage de l'autopilote

Le panneau de la page AFS_CP consacré au paramétrage de l'autopilote permet :

- L'engagement de l'autopilote grâce aux PicturePushButton AP1 et AP2.
- L'affichage du directeur de vol sur le PFD.
- L'engagement du système d'auto-manette.
- Gérer les différents paramètres de l'autopilote :
    - Gestion de la vitesse.
    - Gestion du cap ou de la trajectoire.
    - Gestion de l'altitude.

- Gestion de la vitesse ascensionnelle ou de l'angle de trajectoire.
- Engagement de la capture du localizer.
- Engagement de l'atteinte de l'altitude spécifiée à la vitesse maximale.
- Engagement du système d'auto-manette.

La gestion des *PicturePushButton* LOC, ALT, APPR, FD et A/THR est similaire à celle des *PicturePushButton* LS et VV que nous avons présentés précédemment (trois barres noires affichées lorsque le service correspondant est désactivé et trois barres vertes affichées lorsqu'il est activé), nous ne la présentons donc pas ici.

La gestion des *PicturePushButton* responsables de l'engagement de l'autopilote est similaire mais diffère en un point : il est nécessaire de savoir si l'un des deux autopilotes est engagé. En effet, le panneau de gestion des paramètres de l'autopilote n'est activé que lorsqu'au moins l'un des deux autopilotes est engagé. Le modèle ICO de la gestion d'un tel comportement est présenté en Figure 9.10. La fonction d'activation associée est présentée dans le Tableau 9.12, et la fonction de rendu dans le Tableau 9.13. Ainsi, lorsque l'un des deux autopilotes est activé par un clic sur le *PicturePushButton* correspondant, le jeton de la place AP_OFF est enlevé et un jeton est placé dans la place AP_ON. L'opération inverse sera effectuée si tous les autopilotes sont désactivés.

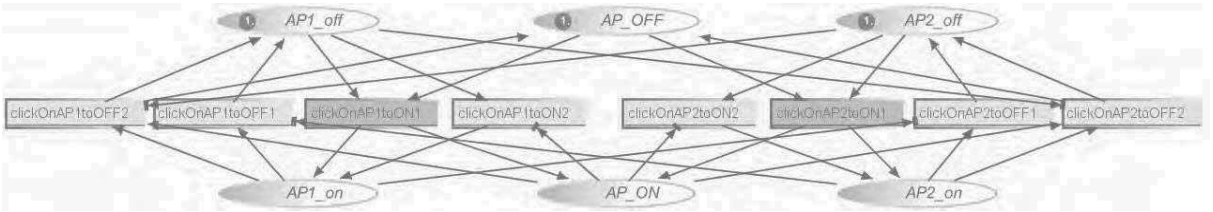


Figure 9.10. Modèle ICO de la gestion de l'engagement de l'autopilote

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_AP1	Clic	clickOnAP1	clickOnAP1toON1, clickOnAP1toON2, clickOnAP1toOFF1, clickOnAP1toOFF2
PPB_AP2	Clic	clickOnAP2	clickOnAP2toON1, clickOnAP2toON2, clickOnAP2toOFF1, clickOnAP2toOFF2

Tableau 9.12 Fonction d'activation pour la gestion de l'engagement de l'autopilote

État	Places	Événement(s)	Rendu associé
Autopilote désengagé	AP_OFF	Jeton entré Marquage réinitialisé	Désactivation du panneau de gestion des paramètres de l'autopilote
Autopilote engagé	AP_ON	Jeton entré	Activation du panneau de gestion des paramètres de l'autopilote
AP1 désengagé	AP1_OFF	Jeton entré Marquage réinitialisé	Affichage des trois barres noires sur PPB_AP1
AP1 engagé	AP1_ON	Jeton entré	Affichage des trois barres vertes sur PPB_AP1
AP2 désengagé	AP2_OFF	Jeton entré Marquage réinitialisé	Affichage des trois barres noires sur PPB_AP2
AP2 engagé	AP1_ON	Jeton entré	Affichage des trois barres vertes sur PPB_AP2

Tableau 9.13. Fonction de rendu pour la gestion de l'engagement de l'autopilote

Lorsque le panneau de gestion des paramètres de l'autopilote est activé, le pilote peut gérer les paramètres de vitesse, de cap, de trajectoire, d'altitude, de vitesse ascensionnelle et d'angle de trajectoire. Chacun de ces paramètres peut être *Managed* ou *Selected*.

Nous ne présentons pas ici les comportements des différents paramètres car ils sont assez similaires et les principes de modélisation mis en jeu sont similaires. Nous prenons l'exemple de la vitesse pour expliquer les principes mis en jeu par ce comportement spécifique. Le modèle ICO de la gestion de la vitesse est présenté en Figure 9.11. La gestion de l'*EditBoxNumeric* est la même que celle que nous avons présentée pour la référence de pression atmosphérique. L'unique différence ici est le fait que la

vitesse soit modifiable en permanence. Nous ne présentons pas les fonctions d'activation et de rendu correspondantes à cette partie du modèle.

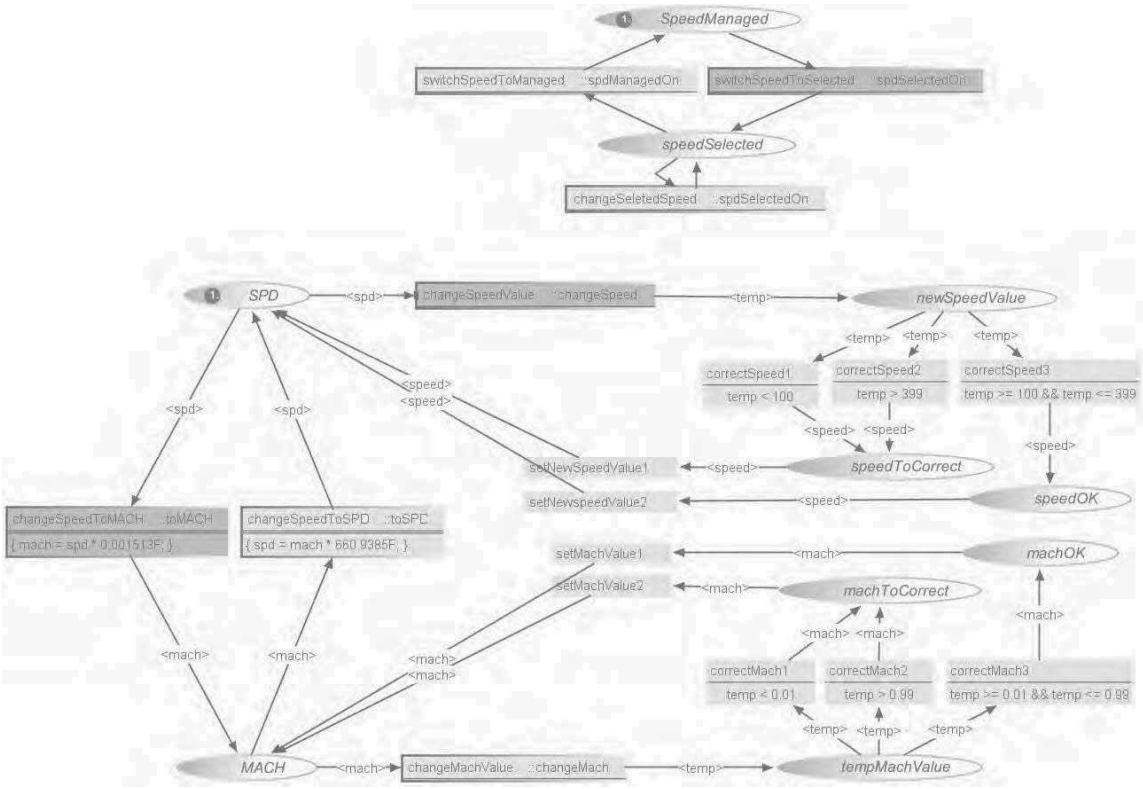


Figure 9.11. Modèle ICO de la gestion de la vitesse

Le mode de gestion de la vitesse (*Managed* ou *Selected*) est géré par le modèle présenté en haut de la Figure 9.11. La fonction d'activation correspondant à cette partie du modèle est présentée dans le Tableau 9.14, la fonction de rendu dans le Tableau 9.15. Lorsque la vitesse est en mode *Managed*, le pilote peut décider d'entrer une nouvelle valeur de vitesse et de passer en mode *Selected* de manière à engager cette nouvelle vitesse. Pour cela, après avoir entré une nouvelle vitesse dans l'*EditBoxNumeric* correspondante, il clique sur le *PicturePushButton* `Selected` correspondant à la vitesse, ce qui déclenche le franchissement de la transition `switchSpeedToSelected`. Dans le mode *Selected*, le pilote peut décider de modifier la vitesse de nouveau. Dans ce cas, après avoir modifié la vitesse dans l'*EditBoxNumeric* correspondante, il peut de nouveau cliquer sur le *PicturePushButton* `Selected` pour engager la nouvelle vitesse (franchissement de la transition `changeSelectedSpeed`). Le pilote peut également décider de repasser en mode *Managed* en cliquant sur le *PicturePushButton* `Managed` correspondant, ce qui entraine le franchissement de la transition `switchSpeedToManaged`.

Les rendus graphiques associés à ces différents modes sont présentés en Figure 9.12.

Objet d'interaction	Action de l'utilisateur	Event handler	Transition(s) du contrôleur de dialogue associées à l'event handler
PPB_Selected	Clic	spdSelectedOn	switchSpeedToSelected, changeSelectedSpeed
PPB_Managed	Clic	spdManagedOn	switchSpeedToManaged

Tableau 9.14 Fonction d'activation pour la gestion de la vitesse

État	Places	Événement(s)	Rendu associé
Vitesse en mode *Managed*	SpeedManaged	Jeton entré Marquage réinitialisé	Affichage de la vitesse en magenta
Vitesse en mode *Selected*	speedSelected	Jeton entré	Affichage de la vitesse en cyan

Tableau 9.15. Fonction de rendu pour la gestion de la vitesse

a) Mode Managed activé | *b) Mode Selected activé*

Figure 9.12. Rendus graphiques associés aux modes de gestion de la vitesse

9.2.2 Modélisation des widgets

Le FCUA regroupe douze types de widgets différents :

- Des objets de regroupement (ou *container*) : *Layer*, *BasicContainer*, *Panel* et *RadioBox*.
- Des widgets d'action : *CheckButton*, *ComboBox*, *EditBoxNumeric*, *PicturePushButton* et *PictureToggleButton*.
- Des widgets d'affichage : *Label*, *GP_Line* et *Picture*.

Nous avons exposé dans la section 5.4.4 les principes génériques pour la modélisation du comportement des widgets à l'aide de la notation ICO. Il est tout d'abord important de déterminer les différents paramètres des différents widgets ainsi que les événements qu'ils sont susceptibles d'envoyer.

À la vue du grand nombre de widgets présents dans l'application, nous choisissons de n'en présenter que trois : l'*EditBoxNumeric*, la *RadioBox* et le *CheckButton*. Ces trois widgets sont particulièrement intéressants car ils apportent de nouvelles problématiques de modélisation :

- L'*EditBoxNumeric* utilise le mode *Caging* du serveur ainsi que les événements provenant des actions utilisateur sur le clavier (pour la saisie de données).
- La *RadioBox* est un container spécial car elle permet de maintenir deux (ou plus) widgets comprenant un état interne (comme par exemple les *CheckButton* qui peuvent être sélectionnés ou non) en exclusivité mutuelle : un seul des widgets contenus dans la *RadioBox* est sélectionné.
- Le *CheckButton* doit pouvoir gérer les appels de méthodes provenant d'une *RadioBox*.

Le Tableau 9.16 présente les différents paramètres de ces widgets ainsi que les événements qu'ils sont susceptibles d'envoyer. La Figure 9.13 présente les modèles CompoNet de ces trois widgets et le Tableau 9.17 présente les erreurs que ces widgets sont susceptibles d'envoyer.

Widget	Interactive	A661_Event	Parameter	
			DesignTime	Runtime
RadioBox	No		WidgetType WidgetID ParentID	Visible Enable
CheckButton	Yes	A661_EVT_STATE_CHANGE	WidgetType WidgetID ParentID PosX PosY SizeX SizeY MaxStringLength Alignment PicturePosition	Visible Enable StyleSet LabelString
EditBoxNumeric	Yes	A661_EVT_STRING_CHANGE_ABORTED A661_EVT_STRING_CHANGE A661_EVT_STRING_CONFIRMED A661_EVT_EDITBOX_OPENED	WidgetType WidgetID ParentID PosX PosY SizeX SizeY MaxFormatStringLength LegendRemoved	Visible Enable StyleSet Value FormatString EntryValidation LegendString LegendPosition MinValue MaxValue

Tableau 9.16. Caractéristiques des widgets de l'application FCUA

Widget	Error	Parameter	Error identifier
RadioBox			
CheckButton	LabelString length > MaxStringLength	LabelString	A661_ERROR_STRING_LENGTH
EditBoxNumeric	Legend String length > MaxStringLength	A661_STRING	A661_ERROR_STRING_LENGTH
	Value is not compliant with the FormatString when FormatString contains one of '+', '_', '#'	A661_VALUE	A661_ERROR_CONFIGURATION
	Value < MinValue	A661_VALUE	A661_ERROR_MIN
	Value > MaxValue	A661_VALUE	A661_ERROR_MAX
	MinValue is not compliant with the FormatString when FormatString contains one of '+', '_', '#'	A661_MINMAX_VALUES	A661_ERROR_CONFIGURATION
	MaxValue is not compliant with the FormatString when FormatString contains one of '+', '_', '#'	A661_MINMAX_VALUES	A661_ERROR_CONFIGURATION

Tableau 9.17. Description des erreurs pour les widgets de l'application FCUA

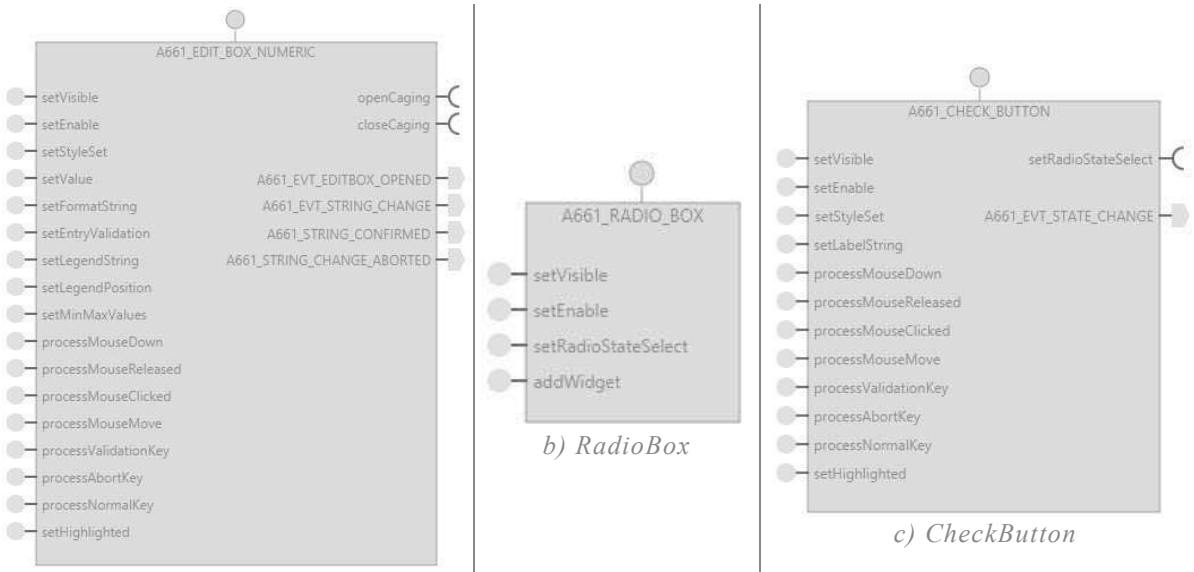


Figure 9.13. Modèles CompoNet des widgets du FCUA que nous présentons

9.2.2.1 Modélisation d'une EditBoxNumeric

La Figure 9.14 présente le modèle ICO d'une *EditBoxNumeric*. Le comportement d'une *EditBoxNumeric* est assez compliqué mais peut être résumé en trois états principaux, chacun représenté par une place du même nom :

- `Idle` correspond à l'état durant lequel l'*EditBoxNumeric* est en attente d'un événement correspondant à une action de l'utilisateur.
- `Editing` correspond à l'état durant lequel l'utilisateur peut saisir une donnée dans l'*EditBoxNumeric* à l'aide du clavier.
- `WaitingForUA` correspond à l'état durant lequel l'*EditBoxNumeric* attend la confirmation de l'UA pour sa nouvelle valeur.

Lorsqu'elle est en mode *Idle* (jeton dans la place `Idle`), l'*EditBoxNumeric* attend un clic de l'utilisateur. Dans ce cas, la réception d'un clic utilisateur (jeton placé dans la place `SIP_processMouseClicked`) déclenche le franchissement de la transition `processMouseClicked2` qui dépose un jeton dans la place `clicked`. Le franchissement de la transition `openCaging` qui suit ce dépôt de jeton permet de demander au serveur d'activer le mode *Caging*. Une fois le mode *Caging* activé, un jeton est placé dans la place `opened`, ce qui a pour effet de déclencher le franchissement de la transition `triggerEditBoxOpened` qui permet de faire passer l'*EditBoxNumeric* dans l'état *Editing* (jeton dans la place `Editing`).

Lorsque l'*EditBoxNumeric* est dans l'état *Editing*, trois scénarios sont possibles :

- L'utilisateur saisit une valeur en tapant sur les touches du clavier, ce qui déclenche des appels de la méthode `processNormalKey` et valide cette valeur en tapant sur la touche de validation, ce qui déclenche l'appel de la méthode `processValidationKey`. Dans ce cas, l'*EditBoxNumeric* passe dans l'état *WaitingForUA* (jeton dans la place `WaintingForUA`) après avoir envoyé l'événement `A661_EVT_STRING_CONFIRMED` à l'UA.
- L'utilisateur saisit une valeur mais décide de l'annuler. Pour cela, il tape la touche d'annulation ce qui a pour effet de déclencher l'appel de méthode `processAbortKey` et de replacer l'*EditBoxNumeric* dans l'état *Idle* avec l'ancienne valeur.
- Le temps limité pour l'édition est dépassé sans que l'utilisateur n'ait annulé ou validé la valeur qu'il est en train de saisir. Dans ce cas, la saisie est ignorée et l'*EditBoxNumeric* passe de nouveau dans l'état *Idle* avec l'ancienne valeur.

Dans les trois cas, l'*EditBoxNumeric* demande au serveur de désactiver le mode *Caging* avant de passer dans l'état suivant.

Lorsqu'elle est dans l'état *WaitingForUA*, l'*EditBoxNumeric* attend la confirmation de l'UA par rapport à la valeur qui a été saisie et qu'elle vient de lui envoyer. Cette confirmation passe par un appel à la méthode `setEntryValidation`. Si l'UA fait un appel à cette méthode avec la valeur `true`, elle confirme la valeur, ce qui a pour effet de déclencher le franchissement de la transition `setEntryValidation_1` qui replace l'*EditBoxNumeric* dans l'état *Idle* avec la valeur venant d'être saisie. Si l'UA fait un appel à cette méthode avec la valeur `false`, elle infirme la valeur, ce qui a pour effet de déclencher le franchissement de la transition `setEntryValidation_2` qui replace l'*EditBoxNumeric* dans l'état *Idle* avec l'ancienne valeur. Enfin, si l'UA ne fait pas appel à cette méthode dans un temps imparti, cela déclenche le franchissement de la transition `timeOut_`, ce qui infirme la valeur saisie et replace l'*EditBoxNumeric* dans l'état *Idle* avec l'ancienne valeur.

La Figure 9.14 étant très difficile à lire, nous proposons en Figure 9.15 un extrait de ce modèle présenté de manière à mettre en évidence le scénario suivant d'utilisation de l'*EditBoxNumeric* :

- Ouverture de l'*EditBoxNumeric* par un clic de l'utilisateur (`MouseClick management`).
- Saisie d'une valeur (`PressNormalKey management`).
- Validation de la saisie (`PressValidationKey management`).
- Validation de cette valeur par l'UA (`UA validation management`).

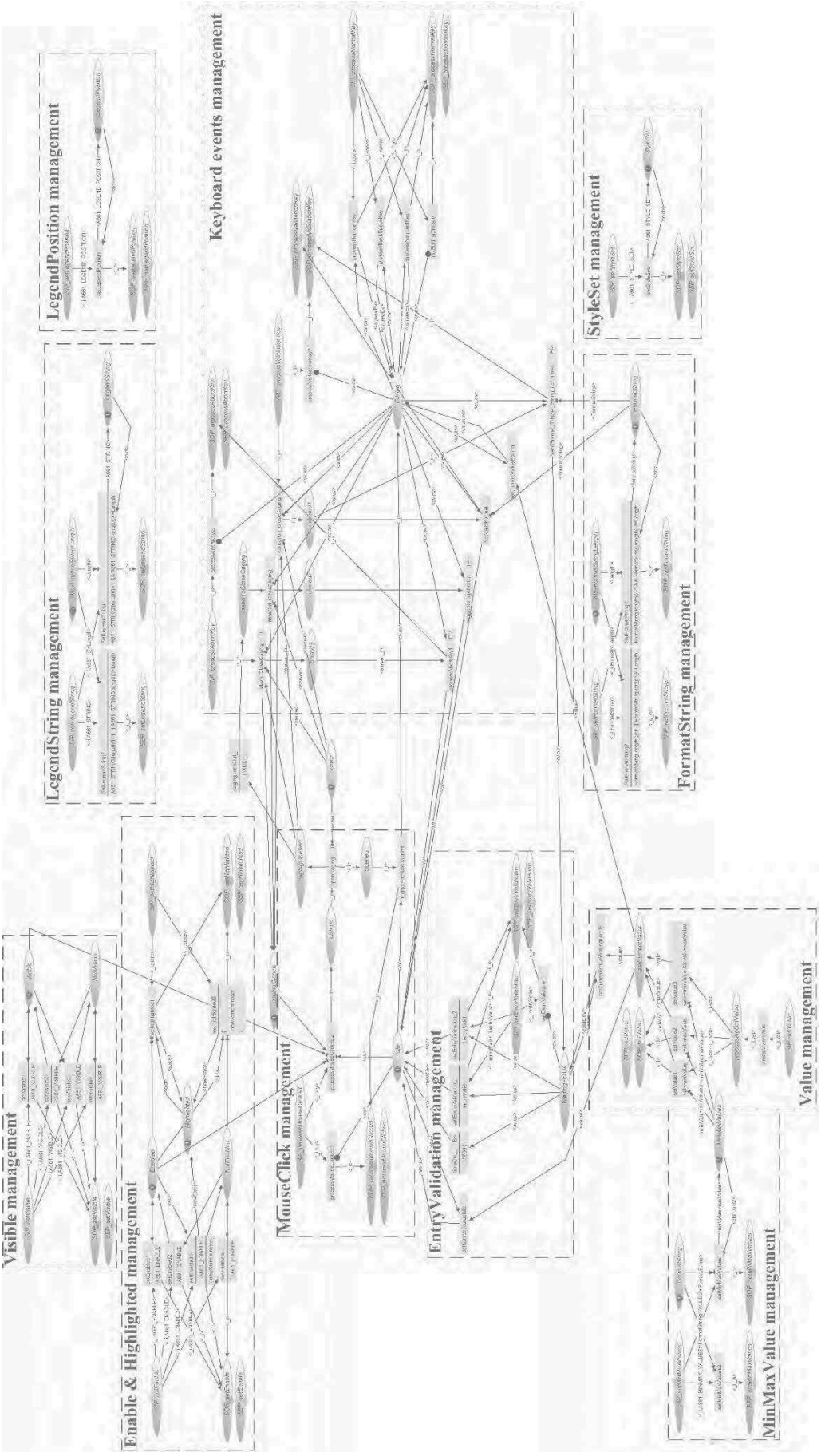


Figure 9.14. Modèle ICO d'une EditBoxNumeric

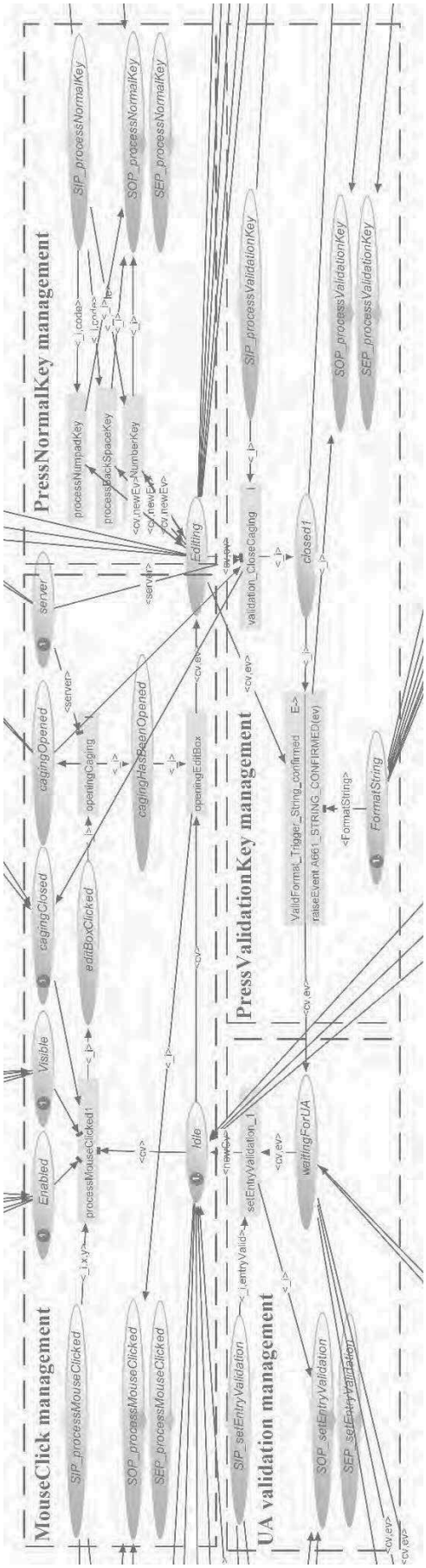


Figure 9.15. Extrait du modèle ICO d'une EditBoxNumeric

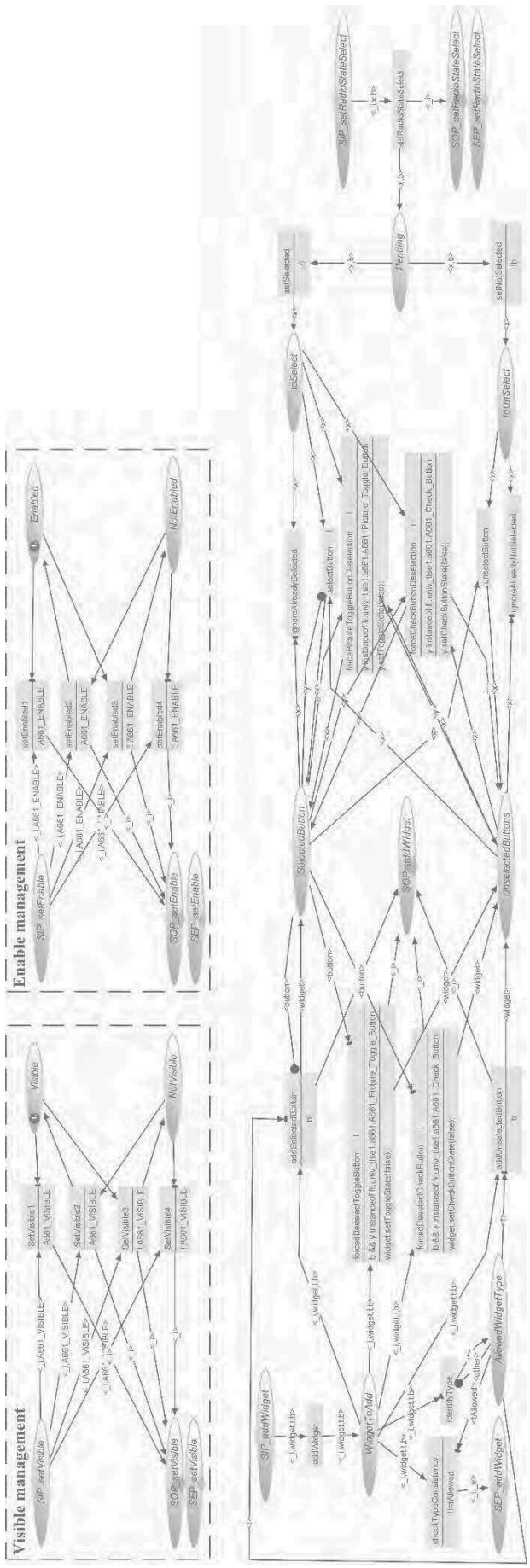


Figure 9.16. Modèle ICO d'une RadioBox

9.2.2.2 Modélisation d'une RadioBox

La Figure 9.16 présente le modèle ICO du comportement d'une *RadioBox*. Celle-ci ne comporte que deux paramètres modifiables en exécution : `Visible` et `Enable`. Cependant, elle présente un comportement particulier afin de permettre de garantir l'exclusivité entre la sélection des widgets qu'elle contient. Il est important de noter qu'une fois sélectionné, un widget ne peut être désélectionné par un clic de l'utilisateur. Cependant, un clic sur un autre des widgets contenus dans la RadioBox a pour effet de désélectionner le widget sélectionné jusqu'alors afin de garantir qu'il n'y ait qu'un seul widget sélectionné.

Pour cela, lors de l'initialisation, le serveur fait appel à la méthode `addWidget` afin de renseigner la *RadioBox* avec les widgets qu'elle contient. Ceci lui permet, lorsque l'un des widgets qu'elle contient la notifie qu'il a été sélectionné, de pouvoir désélectionner tous les autres widgets qu'elle contient. Concrètement, lorsque l'un des widgets qu'elle contient lui notifie qu'il a été sélectionné (appel de la méthode `setRadioSelect` avec la valeur `true`), cela déclenche le franchissement de la transition `setSelected` qui place un jeton dans la place `toSelect`. Dans ce cas, il y a trois possibilités :

- Le widget venant d'être sélectionné était déjà sélectionné. Dans ce cas, la transition `ignoreAlreadySelected` est franchie et cela n'implique aucun changement.
- Aucun widget n'était sélectionné. Dans ce cas, la transition `selectButton` est franchie. Son franchissement garantit la sélection du widget en la forçant avec un appel de méthode et place un jeton contenant le widget sélectionné dans la place `SelectedButton`.
- Un widget différent de celui qui vient d'être sélectionné était sélectionné. Dans ce cas, la transition `forceCheckButtonDeselection` est franchie si le widget sélectionné est un *CheckButton*, ou la transition `forcePictureToggleButtonDeselection` est franchie si le widget sélectionné est un *PictureToggleButton*. Le franchissement de ces transitions a pour effet de forcer une désélection du widget précédemment sélectionné et de le remplacer dans la place `SelectedButton` par le widget venant d'être sélectionné.

9.2.2.3 Modélisation d'un CheckButton

La Figure 9.17 présente le modèle ICO du comportement d'un *CheckButton*. La spécificité de ce widget est qu'il a un comportement différent si son parent direct est un container classique ou une *RadioBox*.

Si son parent direct est un container classique, le *CheckButton* agit comme un bouton poussoir : il est sélectionné sur réception d'un clic s'il était désélectionné et inversement (franchissement des transitions `setToFalse` et `setToTrue`).

Si son parent direct est une *RadioBox*, il ignore le clic lorsqu'il est déjà sélectionné (transition discardClick) et notifie la *RadioBox* lorsqu'il reçoit un clic alors qu'il est désélectionné (jeton dans la place `radioBoxToNotify2` et franchissement de la transition `notifyRadioBoxThatBecameSelected`). Lorsque son parent direct est une *RadioBox*, le *CheckButton* la notifie également des changements d'états réclamés par l'UA (franchissement des transitions `setCheckButtonState4` et `setCheckButtonState5` qui permettent le déclenchement des transitions `notifyRadioBoxThatBecameUnselected` et `notifyRadioBoxThatBecameSelected`).

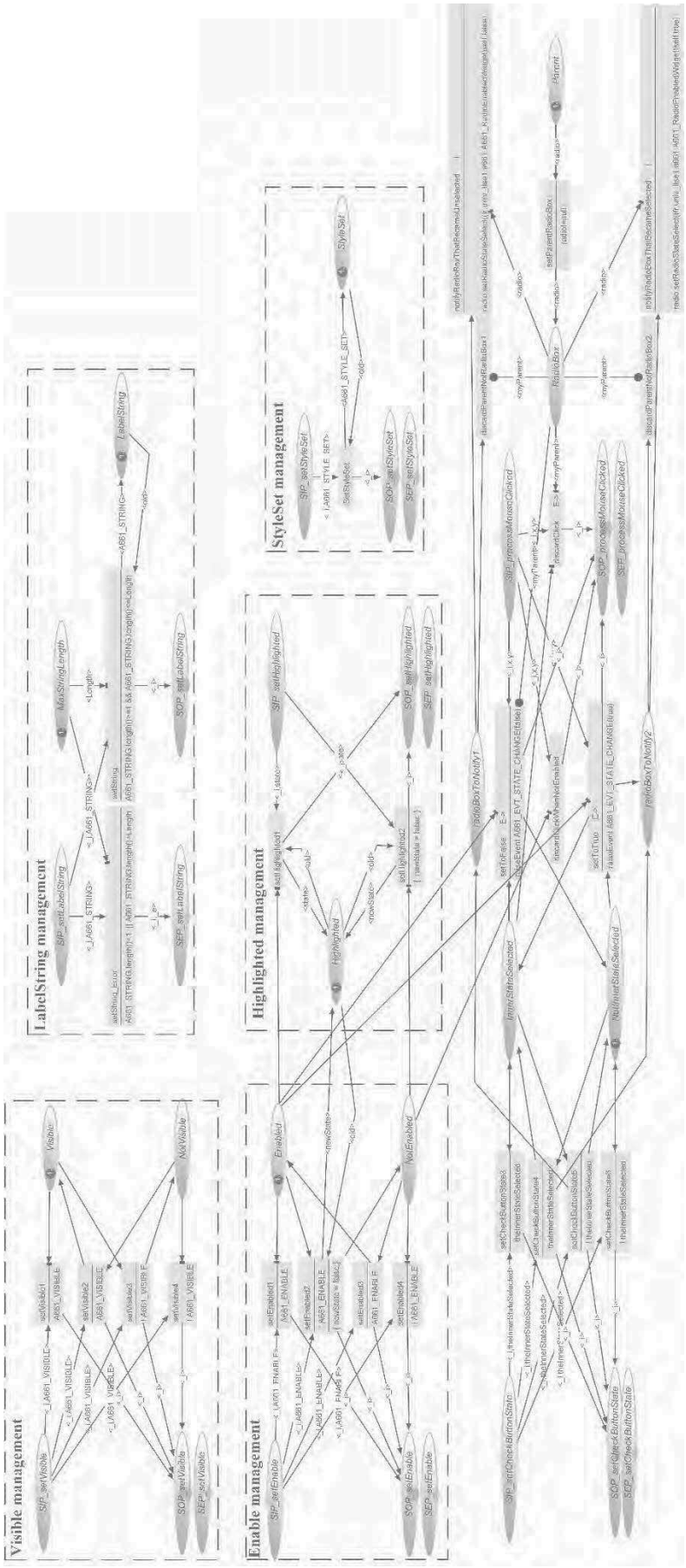


Figure 9.17. Modèle ICO d'un CheckButton

9.2.3 Modélisation du serveur

Nous avons présenté dans la section 5.4.5 les principales fonctionnalités d'un serveur typique des applications avioniques interactives permettant de respecter le standard ARINC 661 : le calcul des techniques d'interaction, le picking et le rendu graphique. Nous ne les présentons donc pas de nouveau dans cette section. Cependant, nous présentons premièrement la gestion du mode *Caging* offert par le serveur et qui permet à certains widgets (tels que l'*EditBoxnumeric* que nous venons de présenter) de bénéficier de ce mode. Deuxièmement, nous présentons le fonctionnement du SceneGraph qui permet de gérer la hiérarchie des widgets.

9.2.3.1 Gestion du mode Caging

Le serveur présente un comportement spécifique pour le curseur : il offre aux widgets un mode d'emprisonnement (appelé `Caging`) qui implique que l'utilisateur ne peux interagir qu'avec le widget ayant demandé ce mode, en utilisant le clavier. Ce mode implique également la disparition du curseur graphique.

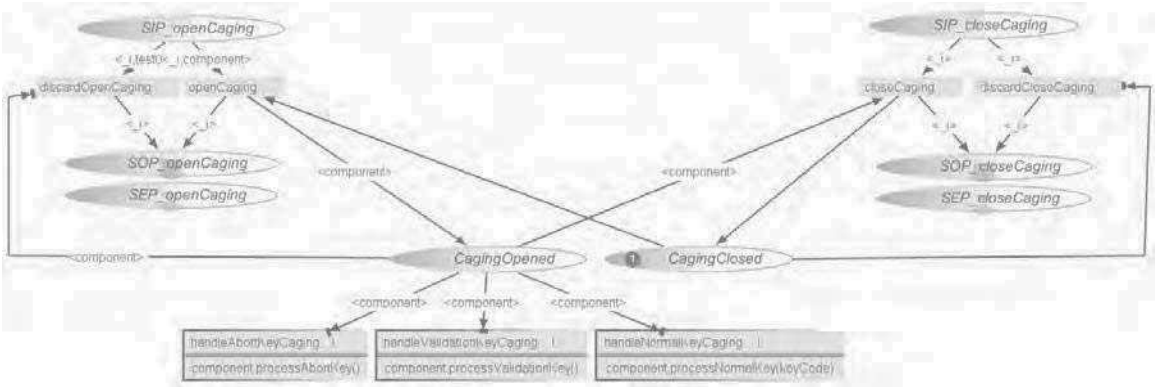


Figure 9.18. Extrait du modèle ICO du serveur : gestion de la fonctionnalité de `Caging`

La modélisation en ICO du comportement du serveur correspondant à cette fonctionnalité est présentée en Figure 9.18. Par défaut, le mode *Caging* du serveur n'est pas activé (jeton dans la place `CagingClosed`). Lorsqu'un widget fait appel à cette fonctionnalité du serveur, un jeton est placé dans la place `SIP_openCaging`. Si le serveur est bien en mode *Caging* désactivé (jeton dans la place `CagingClosed`), il active le mode *Caging* par le franchissement de la transition `openCaging` qui enlève le jeton de la place `CagingClosed` pour le placer dans la place `CagingOpened` et rend le service en plaçant un jeton dans la place `SOP_openCaging`. Si le serveur est déjà en mode *Caging* activé lors de l'appel de méthode, celui-ci est ignoré par le franchissement de la transition `discardOpenCaging` qui n'a aucun impact sur le comportement.

Lorsque le widget le désire, il peut demander au serveur de désactiver de nouveau le mode *Caging*. Pour cela, il fait un appel à la méthode `closeCaging`. Cet appel place un jeton dans la place `SIP_closeCaging`. Le franchissement de la transition `closeCaging` permet alors de désactiver le mode *Caging* en transférant le jeton de la place `CagingOpened` à la place `CagingClosed` et en rendant le service. De la même manière que lors de son activation, si le mode *Caging* est déjà désactivé (jeton dans la place `CagingClosed`), cet appel de méthode est ignoré par le franchissement de la transition `discardCagingClosed`.

Lorsque le mode *Caging* est activé, seul le widget ayant demandé son activation (component) peut recevoir les événements clavier correspondants aux actions de l'utilisateur. La transmission de ces événements est représentée par les trois transitions `handleAbortKeyCaging`, `handleValidationKeyCaging` et `handleNormalKeyCaging` qui sont franchies lors de la réception des événements correspondants.

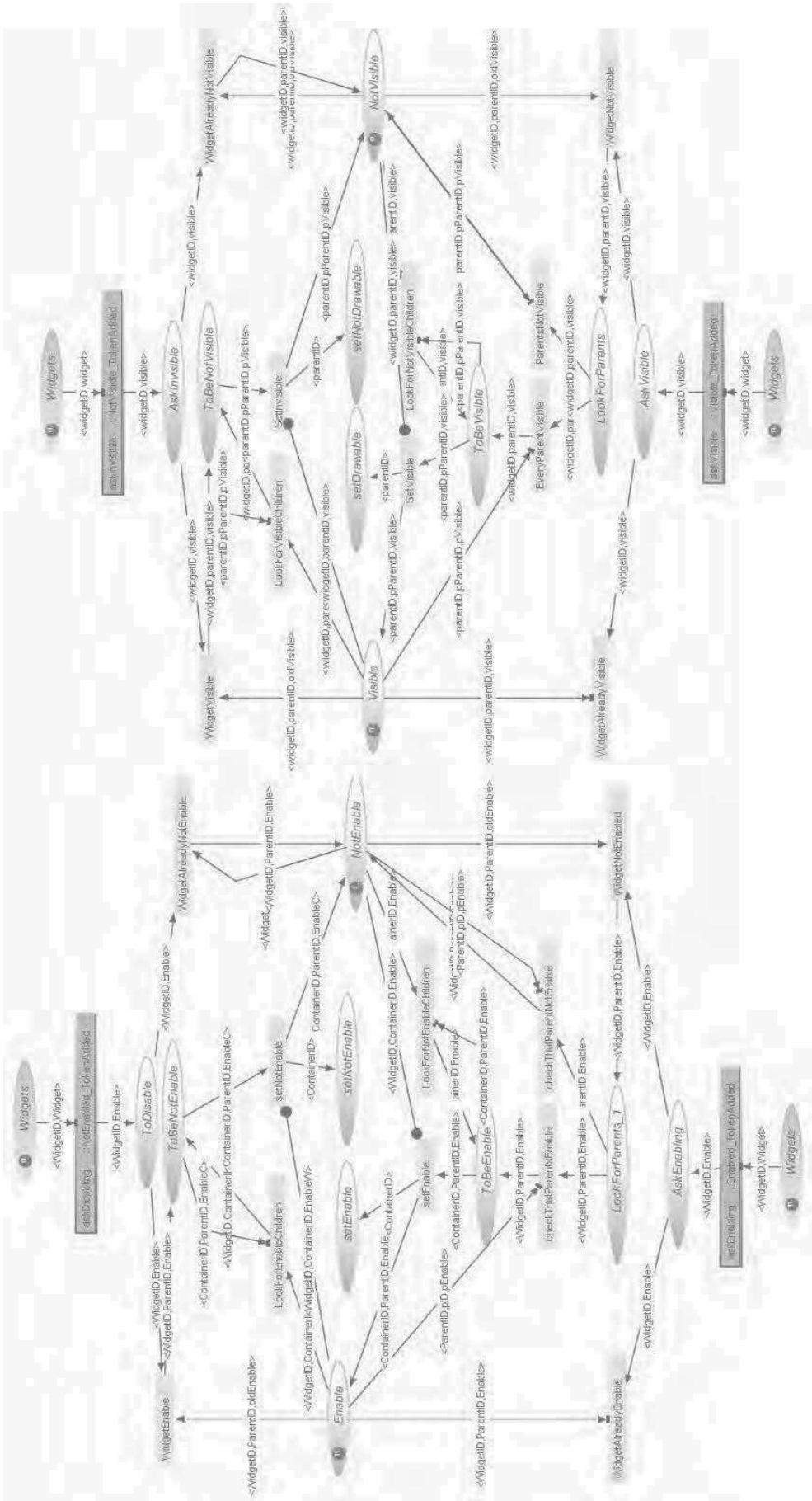


Figure 9.19. Modèle ICO de la gestion des paramètres Enable et Visible par le SceneGraph

9.2.3.2 Gestion de la hiérarchie des widgets

Le *graphe de scène* (ou *SceneGraph*) est responsable de la gestion de la hiérarchie des widgets. Il regroupe toutes les caractéristiques de l'ensemble des widgets nécessaires à la fois au rendu graphique et au picking.

Plus concrètement, il est responsable du calcul de la visibilité et l'activation finale des widgets ainsi que du calcul de leur position absolue sur l'écran (ces trois paramètres dépendent de la valeur des paramètres de leurs parents, voir section 3.1.2.2). Il regroupe également tous les éléments spécifiques pour le rendu graphique de chaque widget. Enfin, il propose une méthode `findWidgetAt` qui permet de déterminer, à partir d'une position (`x`,`y`) sur l'écran donnée, le widget actif et visible situé sous cette position s'il y en a un.

9.2.3.2.1 Gestion des paramètres Visible et Enable

La Figure 9.19 présente le modèle ICO de la gestion du calcul des paramètres `Visible` et `Enable` des widgets. Pour chaque widget, ces deux paramètres dépendent de la valeur des paramètres de leurs parents (voir section 3.1.2.2). Plus concrètement, la visibilité et l'activation des widgets peut être divisée en deux paramètres : le paramètre A661 (qui peut être modifié par l'UA et qui est présent dans chacun des modèles ICO de widgets que nous avons présentés) et le paramètre *contextuel*, qui dépend de la valeur des paramètres A661 de tous ses parents et donc du contexte de ce widget. Ainsi, si l'on prend l'exemple de la visibilité, qu'elle que soit la valeur du paramètre A661 du widget, si l'un des parents du widget a son propre paramètre A661 à `false`, alors la valeur du paramètre contextuel du widget sera à `false`. Concrètement, dans ce cas, il ne sera pas affiché à l'écran et ne pourra donc pas être visible par l'utilisateur.

Pour résumer, la valeur du paramètre A661 est gérée par chacun des widgets séparément, celle du paramètre contextuel est gérée par le *SceneGraph* car c'est lui qui a la connaissance de tous les widgets.

La visibilité et l'activation contextuelles des widgets sont gérées par des modèles similaires qui sont présentés en Figure 9.19. Pour gérer les paramètres contextuels Visible et Enable, le SceneGraph possède deux places pour chacun de ses paramètres soit un total de quatre places :

- La place `Enable` regroupe tous les widgets pour qui la valeur du paramètre contextuel `Enable` est `true`. Chaque widget est représenté par un jeton contenant l'ensemble de paramètres {`WidgetID`, `ParentID`, `Enable`}, où `WidgetID` correspond à l'identifiant du widget, `ParentID` à l'identifiant de son parent le plus proche et `Enable` correspond à la valeur de son paramètre A661 (celui stocké dans le modèle ICO du widget).
- La place `NotEnable` regroupe tous les widgets pour qui la valeur du paramètre contextuel `Enable` est `false`. Chaque widget est également représenté par un jeton contenant l'ensemble des paramètres {`WidgetID`, `ParentID`, `Enable`}.
- La place `Visible` regroupe tous les widgets pour qui la valeur du paramètre contextuel `Visible` est `true`. Chaque widget est représenté par un jeton contenant l'ensemble des paramètres {`WidgetID`, `ParentID`, `Visible`}, où `WidgetID` correspond à l'identifiant du widget, `ParentID` à l'identifiant de son parent le plus proche et `Visible` correspond à la valeur de son paramètre A661 (celui stocké dans le modèle ICO du widget).
- La place `NotVisible` regroupe tous les widgets pour qui la valeur du paramètre contextuel `Visible` est `false`. Chaque widget est également représenté par un jeton contenant l'ensemble de paramètres {`WidgetID`, `ParentID`, `Visible`}.

Le comportement pour la gestion des paramètres contextuels `Visible` et `Enable` est similaire. Pour l'expliquer, nous prenons l'exemple du paramètre `Enable`. Plus particulièrement, nous explicitons le comportement du SceneGraph lorsqu'un des widgets de l'application devient inactif (le comportement lorsqu'un widget devient actif est similaire à celui-ci).

Le *SceneGraph* est abonné aux changements d'états des widgets. Concrètement, ceci est rendu possible grâce à l'utilisation de l'outil PetShop et de son interprète qui permet de fournir des événements

lorsqu'il y a un changement d'état dans un modèle ICO. Ainsi, le *SceneGraph* est abonné aux entrées de jetons dans les places `Enable` et `NotEnable` du modèle ICO de chaque widget possédant ce paramètre. Pour cela, il est abonné aux méta-événements `NotEnabled_TokenAdded` et `Enabled_TokenAdded` de tous les modèles ICO représentant des widgets contenant ce paramètre. Ces méta-événements sont envoyés par l'interprète de PetShop.

Ainsi, lorsqu'un widget actif devient inactif, la transition `askDisabling` est franchie, ce qui place un jeton, contenant l'identifiant du widget ainsi que la nouvelle valeur de son paramètre `Enable` (`false` dans ce cas), dans la place `ToDisable`. À ce moment, deux cas de figure sont envisageables. Premièrement, si le widget se trouvait déjà dans la place `NotEnable` (par exemple, si l'un de ces parents était inactif), la transition `WidgetAlreadyNotEnable` est franchie et remplace le jeton correspondant au widget dans la place `NotEnable` par un nouveau jeton. Ceci permet de garantir que la valeur du paramètre `Enable` de chacun des widgets contenus dans la place `NotEnable` est mise à jour. Deuxièmement, si le widget se trouvait dans la place `Enable`, la transition `WidgetEnable` est franchie, ce qui place un jeton dans la place `ToBeNotEnable`. À ce moment, deux cas de figure sont envisageables. Premièrement, le widget est un container, il faut s'assurer dans ce cas que tous les widgets qu'il contient et qui sont dans la place `Enable` soient transférés dans la place `NotEnable`. Ce transfert est effectué par le franchissement de la transition `LookForEnableChildren`. Deuxièmement, si le widget n'est pas un container ou si plus aucun des widgets qu'il contient n'est stocké dans la place `Enable`, il est déposé dans la place `NotEnable` par le franchissement de la transition `setNotEnable`. Ce traitement est effectué pour tous les widgets qui se trouvent dans la place `ToBeNotEnable` ce qui permet un traitement récursif pour tous les containers.

Lorsqu'un widget passe dans la place `NotEnable` par le franchissement de la transition `setNotEnable`, un jeton est également déposé dans la place `setNotEnable`. Ce jeton permet la mise à jour du paramètre `Enable` pour le rendu graphique. Le comportement qui s'ensuit est expliqué dans la section 9.2.3.2.3 qui explique la gestion des différents paramètres pour le rendu graphique.

9.2.3.2.2 Gestion des paramètres de position et de taille des widgets

Les paramètres de position (`PosX` et `PosY`) ainsi que ceux de taille (`SizeX` et `SizeY`) sont traités de la même manière que les paramètres `Enable` et `Visible` : lorsque l'un des paramètres de position ou de taille est modifié pour un widget, cette modification est répercutée récursivement sur les enfants de ce widget si celui-ci est un container. Nous ne présentons pas ici tous les modèles correspondants à la gestion de ces deux paramètres car ils sont très similaires. Cependant, afin de clarifier la manière dont ils sont traités, nous présentons l'exemple du paramètre `PosX`.

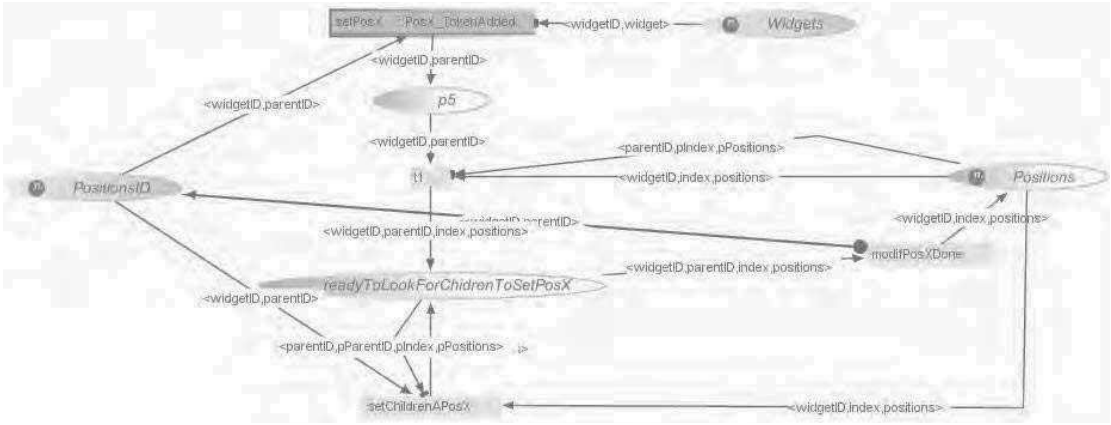


Figure 9.20. Modèle ICO de la gestion des paramètres PosX par le SceneGraph

Le modèle ICO correspondant à la gestion des paramètres `PosX` est présenté en Figure 9.20. Tous les widgets sont stockés dans la place `Positions`. Chaque jeton de cette place correspond à un widget, représenté par l'ensemble des paramètres {`widgetID`, `index`, `positions`}. Lorsque le paramètre `PosX` d'un widget est modifié, la transition `setPosX` est franchie, plaçant un jeton (contenant deux paramètres : `widgetID` l'identifiant du widget et `parentID`, celui de son parent) dans la place `p5`. Ceci entraîne le franchissement de la transition `t1` qui permet de déplacer le jeton correspondant au widget de la place

Positions vers la place readyToLookForChildrenToSetPosX après avoir calculé ses nouvelles coordonnées absolues en fonction de celles de son parent. À ce moment là, si le widget est un container, la transition setChildrenAPosX est franchie pour tous ses enfants, sinon, la transition modifPosDone est franchie, replaçant le widget avec ses nouvelles coordonnées dans la place Positions.

9.2.3.2.3 Gestion des paramètres pour le rendu graphique

Si nous revenons sur le modèle présenté en Figure 9.19, lorsqu'un widget passe dans la place NotEnable par le franchissement de la transition setNotEnable, un jeton est également déposé dans la place setNotEnable. Pour faciliter la lecture du modèle, le traitement du jeton dans cette place est présenté en Figure 9.21 où nous retrouvons la place setNotEnable (sous la forme d'une place virtuelle, voir section 5.2.1). Nous pouvons voir sur ce modèle que le placement d'un jeton dans cette place déclenche le franchissement de l'une des transitions setNotEnable1 ou setNotEnable2. Cette action permet l'actualisation du paramètre Enable du widget dans l'une des places Drawable ou NotDrawable qui sont utilisées pour le rendu graphique. Ces places regroupent en effet tous les widgets de l'application et, pour chaque widget, tous les paramètres nécessaires au rendu graphique.

Ainsi, chaque widget est représenté dans l'une de ces places par un jeton contenant l'ensemble des paramètres suivants {Index, WidgetID, Parameters} où Index correspond à l'index de création du widget, WidgetID à son identifiant et Parameters à l'ensemble des paramètres nécessaires au dessin du widget (par exemple pour un *PicturePushButton* : Enable (contextuel), Highlighted, PictureReference, LabelString et StyleSet). Le paramètre Visible n'est pas inclus dans cette liste car il est représenté par le fait que le jeton correspondant est placé dans la place Drawable (auquel cas Visible=true) ou dans la place NotDrawable (auquel cas Visible=false). La mise à jour de ces paramètres n'est pas représentée ici pour des raisons de lisibilité mais se fait de la même manière que la mise à jour du paramètre Enable : par abonnement au comportement du widget et à la modification de ses paramètres.

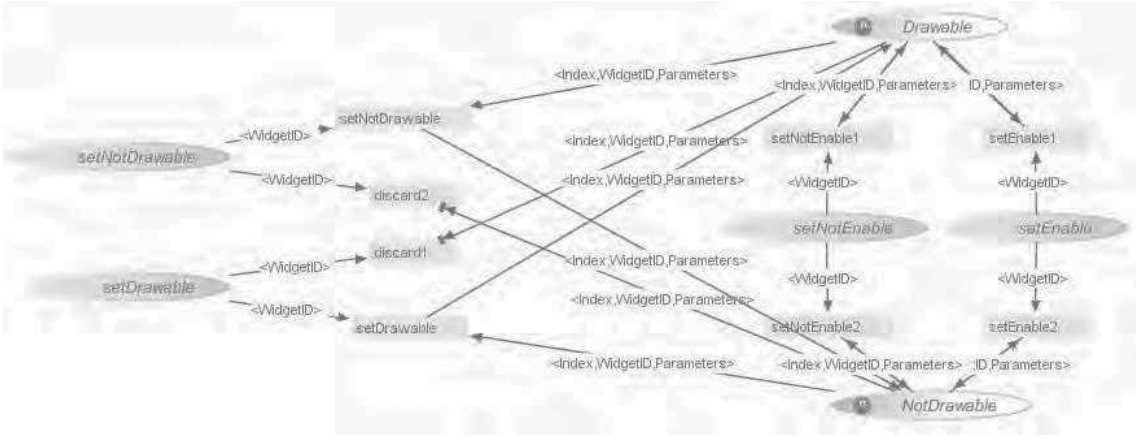


Figure 9.21. Modèle ICO de la gestion des widgets à dessiner par le SceneGraph

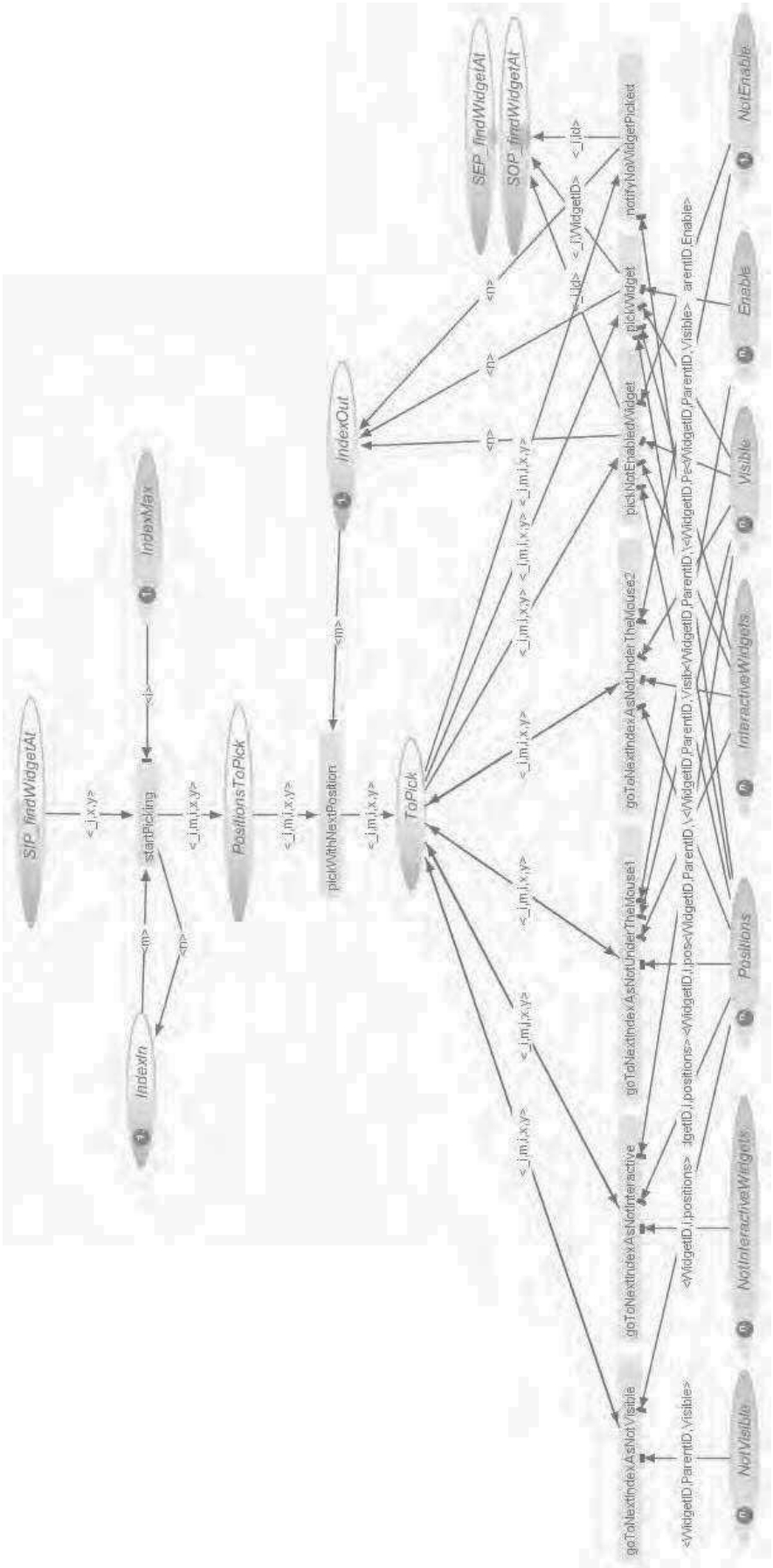


Figure 9.22. Modèle ICO de la gestion de la fonction findWidgetAt du SceneGraph

9.2.3.2.4 Gestion de la fonction findWidgetAt

La fonction `findWidgetAt` permet de déterminer (s'il y en a un) quel est le widget actif et visible à une position donnée. Pour cela, elle parcourt l'ensemble des widgets afin de déterminer le widget concerné. Ce parcours est effectué dans l'ordre inverse du dessin des widgets. En effet, si plusieurs widgets sont superposés au même endroit de l'écran, c'est celui qui est vu par le pilote (et donc, le dernier dessiné) qui est concerné par l'action de celui-ci.

La Figure 9.22 présente le modèle ICO de la gestion de cette fonction. Ainsi, pour chaque appel de cette fonction (jeton placé dans la place `SIP_findWidgetAt`), l'ensemble des widgets est parcouru dans l'ordre inverse de leur dessin. Pour chaque widget, la fonction vérifie cinq cas de figures différents :

- Le widget n'est pas visible (franchissement de la transition `goToNextIndexAsNotVisible`). Dans ce cas, le widget ne peut pas être le widget concerné par l'action de l'utilisateur et on passe au widget suivant.
- Le widget n'est pas un widget interactif (franchissement de la transition `goToNextIndexAsNotInteractive`). Dans ce cas, le widget ne peut pas être le widget concerné par l'action de l'utilisateur et on passe au widget suivant.
- Le widget n'est pas placé sous la position (franchissement de la transition `goToNextIndexAsNotUnderTheMouse1` ou `goToNextIndexAsNotUnderTheMouse2` en fonction de son état actif ou non). Dans ce cas, le widget ne peut pas être le widget concerné par l'action de l'utilisateur et on passe au widget suivant.
- Le widget est placé sous la position mais n'est pas actif (franchissement de la transition `pickNotEnabledWidget`). Dans ce cas, le widget est le widget concerné par le clic mais étant donné que celui-ci est inactif, la fonction retourne `-1`, ce qui signifie qu'aucun widget actif et visible n'est placé sous le curseur.
- Le widget est sous la position et est actif (franchissement de la transition `pickWidget`). Dans ce cas, le widget est celui concerné par l'action de l'utilisateur et la fonction retourne l'identifiant de ce widget.
- Aucun widget visible et actif n'est placé sous la position (franchissement de la transition `notifyNoWidgetPicked`). Dans ce cas, la fonction retourne `-1`.

9.3 Mise en œuvre de l'approche

Tous les modèles que nous venons de présenter ont été développés avec l'outil PetShop (voir section 7.1). Celui-ci offre à la fois un environnement de développement pour les différents modèles ICO et le code Java associé et un environnement d'exécution des modèles.

Nos modèles ICO sont donc des spécifications exécutables du comportement de notre application FCUS. La Figure 9.23 présente la mise en œuvre de l'étude de cas et l'exécution des différents modèles ICO du FCUS à l'aide de l'outil PetShop. Cette figure nous permet de mettre en évidence :

1. Les différentes instances de chaque CO-classe utilisées. Chaque instance d'un type particulier correspond, en exécution, à l'un des 123 widgets du FCUS. Nous pouvons ainsi voir par exemple les douze instances de la CO-classe `A661_BASIC_CONTAINER` qui correspondent aux douze widgets de type *BasicContainer*.
2. Les différents modèles des instances exécutées. Ici nous pouvons voir la partie du contrôleur de dialogue responsable du comportement de la gestion de la vitesse (ici, la vitesse est en mode *Selected*).
3. La fenêtre présentant le FCUS. Nous pouvons voir que le comportement affiché correspond bien à celui présenté dans le point 2 (nous pouvons en effet constater que l'affichage de la vitesse correspond bien au mode *Selected*).

4. La palette d'édition qui permet de glisser/déposer les différents éléments de la notation ICO dans le modèle.
5. Le débogueur.

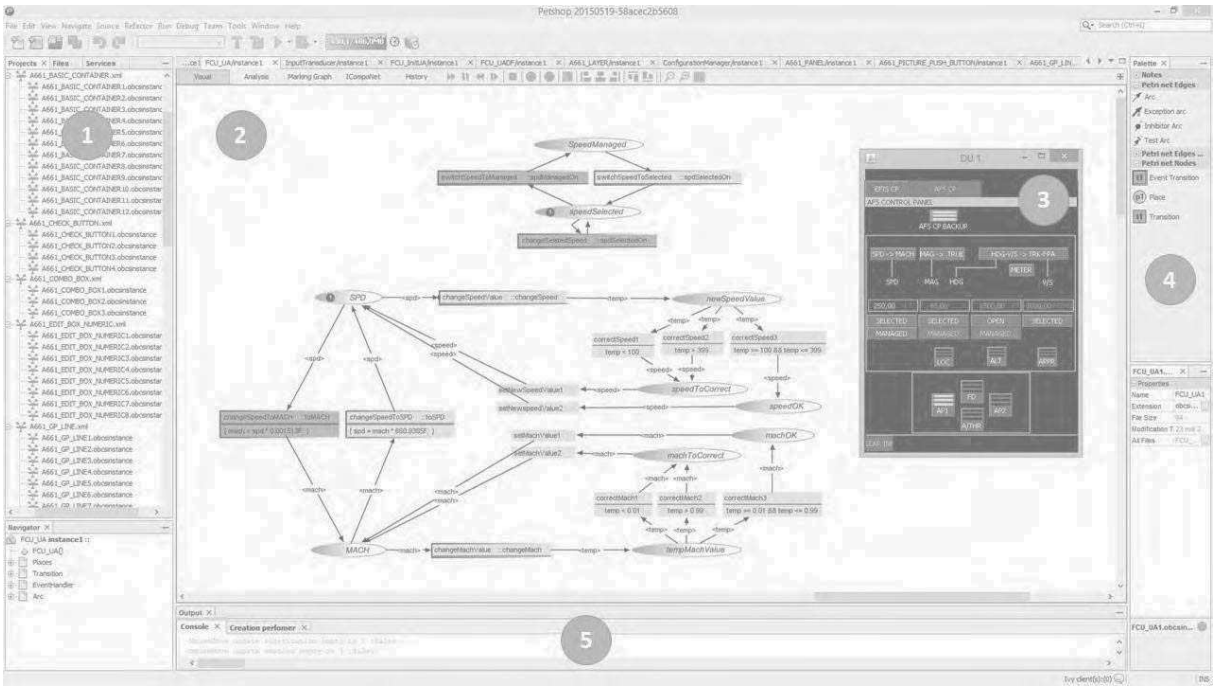


Figure 9.23. Mise en œuvre et éxécution des modèles ICO du FCUS avec l'outil PetShop

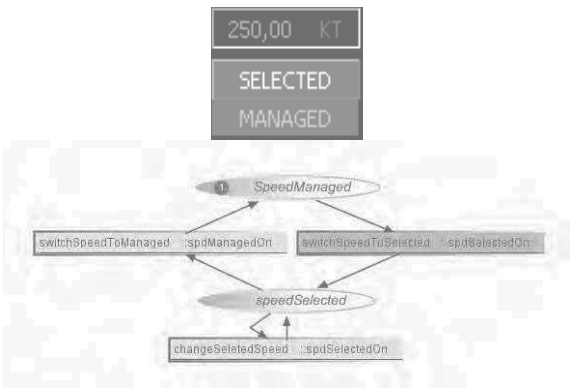
Ainsi, la modélisation de l'application FCUS est constituée de 137 instances de modèles ICO exécutés par l'outil PetShop. Tous ces modèles correspondent aux éléments suivants :

- 123 instances des différents types de CO-classes des widgets correspondent chacunes à la description exécutable du comportement des 123 widgets de l'application.
- 1 instance correspond à la description exécutable du comportement du contrôleur de dialogue.
- 2 instances correspondent à la fonction d'activation et de rendu et 1 instance correspond à la présentation.
- 2 instances correspondent à la description exécutable du comportement du serveur (le SceneGraph et le KidsServer).
- 8 instances sont responsables de l'initialisation de l'application et des différents modèles qui la constituent.

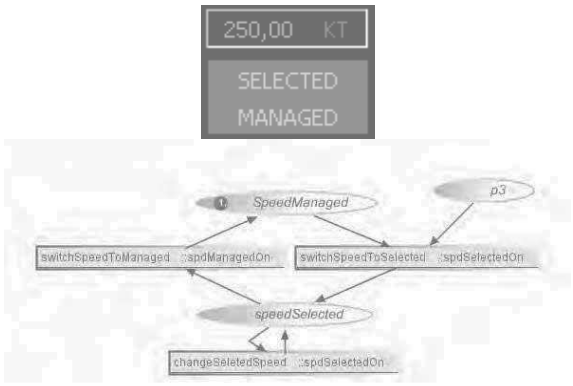
Enfin, étant donné que les différents modèles correspondent aux spécifications exécutables des différents composants logiciels de notre FCUS, il est possible, lors de l'exécution, d'effectuer des modifications sur ces spécifications. Celles-ci ont un impact immédiat sur l'exécution du FCUS.

Pour illustrer ceci, nous prenons l'exemple de la gestion du mode de la vitesse dans le contrôleur de dialogue présentée en section 9.2.1.4. Le comportement normal est présenté en Figure 9.24-a. Dans ce cas, la vitesse est gérée par l'autopilote en mode *Managed* (jeton dans la place `SpeedManaged`) et le pilote peut décider de passer en mode *Selected* en cliquant sur le *PicturePushButton* `Selected`.

Si l'on modifie, durant l'exécution, le modèle en rajoutant une place `p3` inhibant la franchissabilité de la transition `switchSpeedToSelected`. Cette inhibition a pour effet de désactiver l'event handler `spdSelectedOn` qui est associé avec le *PicturePushButton* `Selected` ; elle implique donc également la désactivation de ce bouton (comme montré sur le rendu graphique), le pilote ne peut donc plus alors passer en mode *Selected*.



a) Modèle ICO de la gestion du mode de la vitesse et rendu graphique associé



a) Modèle ICO modifié en exécution de la gestion du mode de la vitesse et rendu graphique associé

Figure 9.24. Illustration de la modification du comportement d'une application durant l'exécution des modèles ICO avec l'outil PetShop

9.4 Synthèse

Dans ce chapitre, nous avons mis en œuvre notre approche à bases de modèles permettant de concevoir des systèmes interactifs zéro-défaut sur l'étude de cas présentée dans le Chapitre 8 : l'application interactive FCUS. Ceci a été effectué en réalisant la modélisation à l'aide de la notation formelle ICO des différents composants logiciels de cette application.

L'utilisation du formalisme ICO nous a permis de décrire de façon complète et non ambigüe le comportement des différents logiciels de notre application. Grâce à l'utilisation de cette notation formelle, nous avons pu décrire :

- Le comportement de l'application FCUS en fonction des événements correspondants aux actions de l'utilisateur sur l'interface. Plus concrètement, celui-ci est décrit à travers l'ensemble des états dans lesquels l'application peut se trouver et comment, pour chacun de ses états, le système peut évoluer (en réagissant à des événements correspondants aux actions de l'utilisateur) vers un autre état.
- Le comportement de l'ensemble des widgets de l'application FCUS. Celui-ci est décrit à travers l'ensemble des états pouvant être atteints par les différents widgets ainsi que les différents changements d'états rendus possibles par la réception d'événements correspondants à des actions de l'utilisateur ou par la réception de commandes provenant de l'UA.
- Le comportement du serveur qui permet le traitement des différents événements provenant de l'utilisateur ainsi que la gestion de la hiérarchie de l'ensemble des widgets. Celui-ci est également décrit à travers l'ensemble des états qu'il peut atteindre et les différents changements d'états rendus possibles par la réception d'événements correspondants à des actions de l'utilisateur ou par la réception de commandes provenant de l'UA.

L'application de notre approche à bases de modèles à cette étude de cas nous permet d'illustrer la faisabilité de la description formelle des différents composants logiciels du système interactif à l'aide de la notation ICO à travers l'exemple d'une étude de cas réaliste inspirée d'un système de commande et contrôle critique.

Chapitre 10. Mise en œuvre de l'architecture logicielle tolérante aux fautes

Sommaire

Afin de valider la faisabilité de notre approche, ce chapitre présente l'application de notre architecture logicielle (présentée dans le Chapitre 6) ainsi que sa mise en œuvre sur l'étude de cas FCUS (présentée dans le Chapitre 8). Ainsi, après avoir défini et analysé le système au travers de son architecture et de diagrammes de séquence, nous identifions ses modes de défaillances. Ceux-ci nous permettent de définir les assertions ainsi que leurs contrôleurs qui nous permettrons d'instancier le composant MON de notre architecture. Enfin, nous présentons la mise en œuvre du FCUS tolérant aux fautes sur un simulateur de système d'exploitation avionique.

La première section présente en détail le système que nous étudions au travers son architecture logicielle ainsi qu'un diagramme de séquence présentant un scénario d'utilisation particulier.

La deuxième section présente l'analyse des modes de défaillance du système pratiquée sur ce diagramme de séquence à travers l'utilisation d'une AMDEC (Analyse des Modes de Défaillance, de leur Effets et de leur Criticité).

La troisième section présente les assertions et leurs contrôleurs que nous avons définis à partir de l'AMDEC présentée en deuxième section.

Enfin, la quatrième section présente la mise en œuvre de notre architecture logicielle tolérante aux fautes appliquée au FCUS.

10.1 Définition et analyse du système

Pour la définition du système, nous nous appuyons sur l'architecture que nous avons définie dans le Chapitre 9. En effet, comme nous l'avons montré dans le Chapitre 7, c'est la version zéro-défaut de notre système modélisé en ICO qui nous sert de composant COM lors de la mise en œuvre de l'architecture tolérante aux fautes.

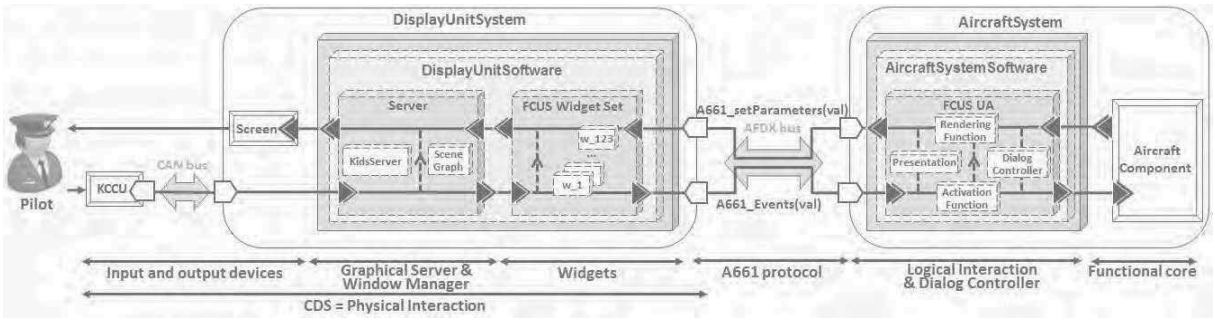


Figure 10.1. Application FCUS dans l'architecture logicielle d'un cockpit interactif

L'architecture logicielle de notre système est présentée en Figure 10.1. Cette représentation haut niveau du système nous permet de mettre en évidence les différents composants que nous utilisons pour effectuer l'analyse du système :

- *Les périphériques d'entrée et de sortie :* le `KCCU` et l'`écran`, qui permettent l'interaction entre le système et l'équipage.
- *Le serveur :* il est composé du `KidsServer` et du `SceneGraph`. Le `SceneGraph` est responsable de la hiérarchie des widgets et le `KidsServer` de toutes les autres fonctionnalités du serveur (calcul des techniques d'interaction, picking, …).
- *Les widgets :* ce sont les composants interactifs de base. Ils correspondent aux éléments d'affichage et d'interaction pour l'application FCUS. Ainsi, ils permettent aux pilotes de déclencher des actions sur les systèmes avioniques et ils permettent à l'UA de notifier aux pilotes les actions qu'ils peuvent effectuer.
- *L'User Application (UA) :* c'est l'interface logicielle des systèmes avioniques. Elle permet de traiter les actions de l'équipage sur les widgets à travers la réception des événements qu'ils envoient. L'UA est également responsable de notifier l'équipage d'un changement d'état du système avionique à travers des appels de méthode pour modifier l'aspect graphique et l'état des widgets. Elle est composée de quatre composants : la présentation, les fonctions d'activation et de rendu et le contrôleur de dialogue.
- *Les systèmes avioniques :* ce sont les systèmes physiques de l'avion. Dans notre cas, il s'agit de l'autopilote ainsi que des logiciels responsables pour l'affichage des écrans PFD et ND.

Pour effectuer l'analyse de ce système, nous nous appuyons sur des scénarios d'utilisation de celui-ci. Ces scénarios sont explicités par des diagrammes de séquence sur lesquels nous pratiquons l'analyse des modes de défaillance. Chaque tâche du pilote (par exemple, engager un nouveau niveau de vol) est associée à un scénario. Étant donné le nombre conséquent de ces scénarios, nous avons choisi d'appliquer dans un premier temps notre approche sur l'un d'entre eux afin d'exemplifier le processus présenté dans ces travaux. L'étude exhaustive des différents scénarios revet en effet plus d'un travail d'ingénierie difficile à effectuer dans le cadre d'une thèse que d'un travail de preuve de concept et de preuve de faisabilité de l'approche qui était notre but dans cette étude de cas.

Nous avons choisi pour cela le scénario de l'engagement d'une nouvelle vitesse dans l'autopilote que nous décrivons ci-après.

Nous considérons ici que le système est dans la configuration suivante : la page AFS_CP est affichée et l'autopilote est engagé ; la vitesse est gérée en mode *Managed*. Dans ce cas, les tâches à réaliser par le pilote pour engager une nouvelle vitesse sont décrites en Figure 10.2. Ce modèle utilise la notation HAMSTERS que nous avons présentée en section 7.3.3.2.

Les tâches effectuées par le pilote sont divisées en deux actions principales :

- *Saisie d'une nouvelle valeur pour la vitesse* (tâche `Enter new value`).

  Premièrement, le pilote place le curseur sur l'*EditBoxNumeric* (EBN) permettant l'édition de la vitesse (appelée `SpeedEBN`) et clique dessus (tâches `Move cursor to SpeedEBN` et `Click on SpeedEBN`). Ces tâches sont abstraites car elles sont assez compliquées et peuvent être développées en tâches élémentaires. Par exemple, le clic sur l'EditBoxNumeric déclenche une mise à jour du côté système : le mode Caging du serveur est activé et le curseur disparait, le pilote ne peut alors interagir qu'avec l'*EditBoxNumeric*. Le pilote saisit ensuite, à l'aide du clavier, la nouvelle valeur de la vitesse et la valide (tâches `Type value`, `Press validation key` et `Input value`). Enfin, quand cette valeur est prise en compte par le système (tâches `Update value` et `Display value`), le pilote vérifie qu'elle est correcte (tâches `Perceive value` et `Analyse that value is OK`).

- *Engagement de cette valeur* (tâche `Engage value`).

  Deuxièmement, le pilote place le curseur sur le *PicturePushButton* (PPB) permettant l'engagement de la vitesse (appelé `SpeedSelectedPPB`) et clique dessus (tâches `Move cusror to SpeedSelectedPPB` et `Click on SpeedSelectedPPB`). Le système engage alors la nouvelle valeur de vitesse et le notifie au

pilote par un changement de la couleur d'affichage de l'*EditBoxNumeric* `SpeedEBN` (tâches `Update state` et `Display value in magenta`). Le pilote vérifie alors que la nouvelle valeur de vitesse a bien été engagée (tâches `Perceive value and color` et `Analyse that new speed has been engaged`).

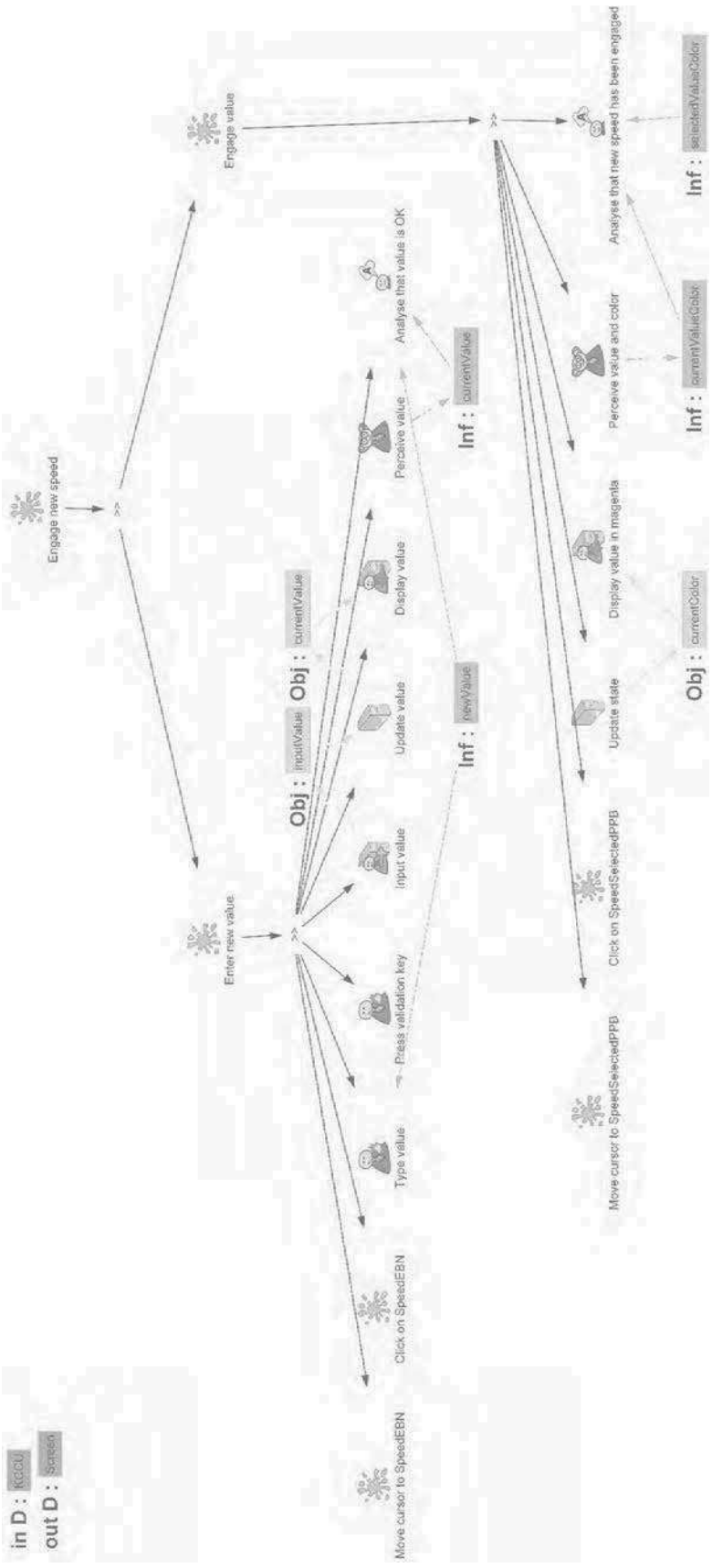


Figure 10.2. Modèle de tâches correspondant au scénario étudié

Le scénario présenté précédemment nous amène au diagramme de séquence présenté en Figure 10.3. Nous rappelons la configuration initiale : la page AFS_CP est affichée et l'autopilote est engagé ; la vitesse est gérée en mode *Managed*. Ce diagramme de séquence ne présente que les composants qui nous intéressent pour ce scénario. Ainsi, il présente :

- Pour le server les composants `KidsServer` et `SceneGraph`.
- Pour les widgets, les deux widgets qui nous intéressent : l'*EditBoxNumeric* permettant l'édition de la vitesse (appelée `SpeedEBN`) et le *PicturePushButton* permettant d'engager la nouvelle vitesse (appelé `SpeedSelectedPPB`).
- Pour l'UA, nous ne présentons qu'un unique composant représentant l'UA correspondant à l'application FCUS. Ce composant contient les différents composants suivants : présentation, fonctions de rendu et d'activation et contrôleur de dialogue. En effet, nous nous intéressons plus particulièrement (comme nous l'avons expliqué dans le Chapitre 6) à la sûreté de fonctionnement des composants génériques des systèmes interactifs ARINC 661, autrement dit, le serveur et les widgets.
- Afin de simplifier ce diagramme, nous avons choisi de ne pas présenter les systèmes avioniques.

Afin d'expliciter ce diagramme de séquence, nous l'avons divisé en dix sections :

1. Le pilote déplace le curseur pour le placer au-dessus de l'*EditBoxNumeric* `SpeedEBN`. À chaque réception d'un événement `mouseMove(x,y)`, le `KidsServer` appelle la fonction `findWidgetAt(x,y)` sur le `SceneGraph` jusqu'à ce que celui-ci renvoie un identifiant `id` différent de `-1`. À ce moment, le curseur est placé au dessus du widget `SpeedEBN` et l'identifiant renvoyé par le `SceneGraph` correspond à l'identifiant du widget `SpeedEBN`, le `KidsServer` appelle alors la fonction `setHighlighted(true)` sur le widget `SpeedEBN` afin de lui donner le focus.
2. Une fois que le curseur est placé sur le widget `SpeedEBN`, le pilote clique sur celui-ci. Ce clic est réceptionné par le `KidsServer` sous la forme d'un événement `mouseClicked(x,y)` qui est transféré au widget `SpeedEBN` qui demande alors au `KidsServer` d'activer le mode *Caging* et passe en mode *Editing* (voir section 9.2.2.1).
3. Le pilote saisit la valeur souhaitée en pressant les touches correspondantes du clavier. Chaque pression de touche est réceptionnée par le `KidsServer` sous la forme d'un événement `pressNormalKey(code)` qui est transféré au widget `SpeedEBN` qui met à jour la valeur en cours d'édition.
4. Le pilote valide la valeur saisie en pressant la touche de validation. Cette action est réceptionnée par le `KidsServer` sous la forme d'un événement `pressValidationKey` qui est transféré au widget `SpeedEBN` qui envoie l'événement `A661_STRING_CONFIRMED(value)` à l'UA `FCUS UA` et passe en mode *WaitingForUA*.
5. L'UA réceptionne cet événement, le transfère aux systèmes avioniques et valide la valeur en appelant la fonction `setEntryValidation(true)` sur le widget `SpeedEBN`.
6. Le widget `SpeedEBN` passe alors en mode *Idle* de nouveau avec la nouvelle valeur de vitesse.
7. Le pilote déplace de nouveau le curseur pour le placer sur le *PicturePushButton* `SpeedSeletectedPPB`. Quand le curseur sort du widget `SpeedEBN`, le `KidsServer` appelle la fonction `setHighlighted(false)` sur celui-ci et quand le curseur passe au-dessus du widget `SpeedSelectedPPB`, le `KidsServer` appelle la fonction `setHighlighted(true)` sur celui-ci pour lui donner le focus.
8. Le pilote clique sur le widget `SpeedSelected`. L'événement `mouseClicked(x,y)` est transféré par le `KidsServer` au widget `SpeedSelected` qui le traite en envoyant l'événement `A661_EVT_SELECTION` à l'UA. Celle-ci notifie l'autopilote qui engage alors la nouvelle valeur de vitesse. L'UA notifie le pilote de l'engagement de la vitesse en appelant la fonction `setStyleSet(selectedStyle)` sur le widget `SpeedEBN`.
9. Le widget `SpeedEBN` traite l'appel de fonction `setStyleSet(selectedStyle)` et met à jour son affichage avec une couleur cyan pour la valeur qu'il affiche.

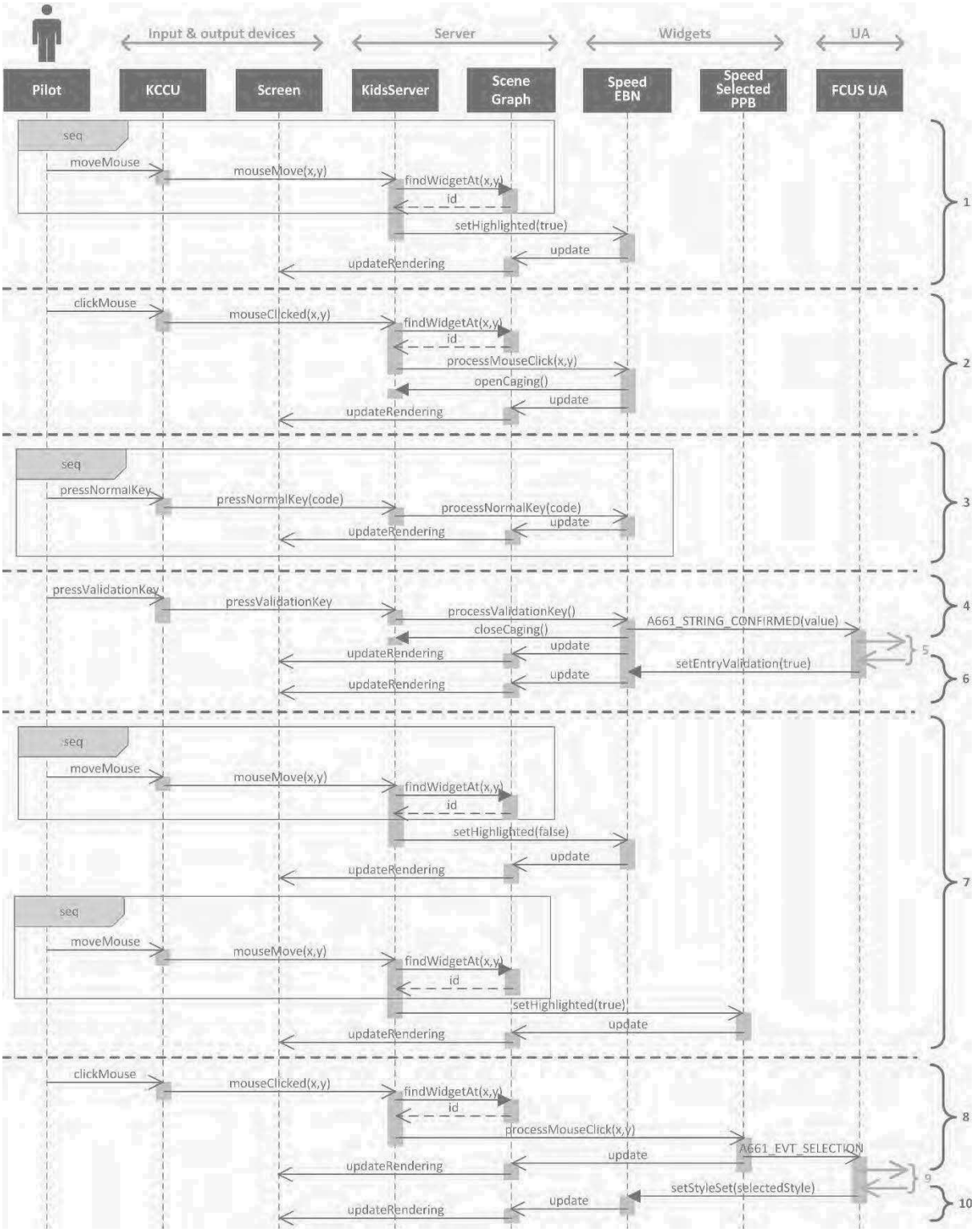


Figure 10.3. Diagramme de séquence de l'engagement d'une nouvelle vitesse

10.2 Identification des modes de défaillance

L'analyse des modes de défaillance du système correspond à l'analyse des modes de défaillances de tous les composants génériques utilisés dans le scénario que nous venons de présenter : le `KidsServer`, le `SceneGraph`, les widgets `SpeedEBN` et `SpeedSelectedPPB`.

Chaque AMDEC est construit de la manière suivante : pour chaque composant, toutes les fonctions utilisées dans le scénario que nous venons de présenter correspondent aux différents items (colonne 1). Pour chacun de ces items, nous avons identifié les trois modes de défaillances suivants (colonne 2) : pas d'exécution de la fonction, mauvaise exécution de la fonction et exécution autonome de la fonction. Comme nous l'avons expliqué dans le Chapitre 6, nous ne présentons pas ici les causes potentielles des défaillances (colonne 3) car nous considérons que celles-ci correspondent à notre modèle de fautes, c'est-à-dire, les fautes logicielles de développement et les fautes matérielles en opération (voir section 4.2.3). Les colonnes 4 et 5 présentent les effets locaux (au niveau de la fonction) et de plus haut niveau (au niveau de ce qui est perçu par le pilote) de chaque mode de défaillance. Enfin, la colonne 6 présente la classification de la défaillance par rapport à la classification que nous avons proposée en section 4.2.2 :

- *Perte de contrôle* : le contrôle demandé par l'utilisateur n'est pas effectué.
- *Contrôle erroné* : le contrôle effectué ne correspond pas à celui demandé par l'utilisateur.
- *Perte d'affichage* : l'affichage correspondant au changement d'état du système n'est pas effectué.
- *Affichage erroné* : l'affichage correspondant à l'état du système est effectué de manière inadéquate.

Identification des modes de défaillance pour les widgets

Le Tableau 10.1 présente un extrait de l'AMDEC que nous avons définie dans le Chapitre 6 (Tableau 6.4). Cet extrait correspond aux défaillances génériques pouvant affecter les widgets et met en évidence les deux types d'items les plus courants pour ceux-ci :

- La gestion des événements résultant des actions de l'utilisateur (`inputProcessing`).
- La gestion des demandes de modification de paramètres (`setParameterProcessing`).

Le deuxième type d'item (`setParamaterProcessing`) concerne tous les widgets possédant des paramètres modifiables en exécution alors que le premier type d'item (`inputProcessing`) ne concerne que les widgets interactifs susceptibles de devoir traiter des événements résultant des actions de l'utilisateur.

L'étude des modes de défaillance des fonctions des deux widgets intervenant dans notre scénario nous a mené à la création du Tableau 10.2 et du Tableau 10.3. Ces deux tableaux instancient le Tableau 10.1 pour les fonctions du *PicturePushButton* `SpeedSelectedPPB` (`ppb` dans le Tableau 10.2) ainsi que pour les fonctions de l'*EditBoxNumeric* `SpeedEBN` (`ebn` dans le Tableau 10.3).

Ainsi, dans notre scénario, l'*EditBoxNumeric* `SpeedEBN` utilise trois items de type `inputProcessing` (`processMouseClicked`, `processNormalKey`, `processValidationKey`), deux items de type `setParameterProcessing` (`setStyleSet` et `setHighlighted`) ainsi qu'un item spécifique (`setEntryValidation`) qui ne permet pas la modification d'un paramètre en particulier mais permet la validation (par l'UA) de la valeur éditée (voir la section 9.2.2.1 expliquant le comportement d'une *EditBoxNumeric*).

Le *PicturePushButton* quant à lui n'utilise ici qu'un seul item de type `inputProcessing` (`processMouseClicked`) et un seul item de type `setParameterProcessing` (`setHighlighted`).

1	2	3	4	5	6
Item	**Failures modes**		**Local effects**	**Upper-level effects**	**Consequence classification**
Widget.inputProcessing Process the click, send the corresponding A661_Event(val) to the UA and send an update notification to the server	No execution Widget.inputProcessing.FM1		Upon the receipt of a kccu input event, the widget does not send any A661_Event(val)	The pilot command is not sent to the aicraft system	Loss of control
	Erroneous excecution Widget.inputProcessing.FM2		Upon the receipt of a kccu input event, the widget sends a wrong A661_Event(val) (or with a wrong value)	A wrong command is sent to the aircraft system	Erroneous control
	Unexpected execution Widget.inputProcessing.FM3		The widget sends an A661_Event(val) without receiving any kccu input event	A command is sent t the aircraft system without any user action	Erroneous control
Widget.setParameterProcessing Process A661_setParameter(val) and send an update notification to the server	No excecution Widget.setParameterProcessing.FM1		The widget does not update its corresponding parameter	The pilot is not notified of the aicraft system state change	Loss of data display
	Erroneous execution Widget.setParameterProcessing.FM2		The widget updates its corresponding parameter with a wrong value	The pilot is notified of a wrong aicraft system state change	Erroneous data display
	Unexpected execution Widget.setParameterProcessing.FM3		The widget updates a parameter without receiving an A661_setParameter	The pilot is notified of an aircraft system state change that did not occur	Erroneous data display

Tableau 10.1. AMDEC résumant les défaillances génériques des widgets

1	2	3	4	5	6
Item	**Failures modes**		**Local effects**	**Upper-level effects**	**Consequence classification**
ppb.processMouseClicked Process the click, send the corresponding A661_EVT_SELECTION to the UA and send an update notification to the server	No excecution ppb.processMouseClicked.FM1		Upon the receipt of a kccu input event, the widget does not send any A661_EVT_SELECTION	The pilot command is not sent to the aicraft system	Loss of control
	Erroneous execution ppb.processMouseClicked.FM2		Upon the receipt of a kccu input event, the widget sends a wrong A661_Event(val)	A wrong command is sent to the aircraft system	Erroneous control
	Unexpected execution ppb.processMouseClicked.FM3		The widget sends an A661_EVT_SELECTION without receiving any kccu input event	A command is sent t the aircraft system without any user action	Erroneous control
ppb.setHighlighted Update the Highlited parameter with the desired value and send an update notification to the server	No execution ppb.setHighlighted.FM1		The widget does not update its Highlighted parameter	The widget stay highlighted (resp. not highlighted) when it should have been not highlighted (reps. highlighted)	Loss of data display
	Erroneous execution ppb.setHighlighted.FM2		The widget updates its Highlighted parameter with a wrong value	The widget stay highlighted (resp. not highlighted) when it should have been not highlighted (reps. highlighted)	Erroneous data display
	Unexpected execution ppb.setHighlighted.FM3		The widget updates its Highlighted parameter without receiving an setHighlighted request	The widget change to highlighted (resp. not highlighted) when it should have stayed not highlighted (resp. highlighted)	Erroneous data display

Tableau 10.2. AMDEC résumant les défaillances du *PicturePushButton* intervenant dans notre scénario

1	2	3	4	5	6
Item	**Failures modes**		**Local effects**	**Upper-level effects**	**Consequence classification**
ebn.processMouseClicked Ask the server to activate the *caging* mode and change mode to *Editing*	No execution ebn.processMouseClicked.FM1		The ebn does not change mode to *Editing*	The pilot can't edit the value	Loss of control
	Erroneous execution ebn.processMouseClicked.FM2		The ebn changes mode to another one than *Editing*	The pilot can't edit the value	Loss of control
	Unexpected execution ebn.processMouseClicked.FM3		The ebn changes mode to *Editing* without any user action	The pilote can't do any actions but editing the value in ebn without asking for	Loss of control
ebn.processNormalKey Process the event and modify the editing value	No execution ebn.processNormalKey.FM1		The ebn does not update the editing value	The last digit typed by the pilot is not taken into account	Loss of control
	Erroneous execution ebn.processNormalKey.FM2		The ebn updates the editing value with a wrong digit	The last digit typed by the pilot is not the one taken into account	Erroneous control
	Unexpected execution ebn.processNormalKey.FM3		The ebn updates the editing value without any user action	The last digit typed by the pilot is not the one taken into account	Erroneous control
ebn.processValidationKey Sends an A661_STRING_CONFIRMED(val) event to the UA and change mode to *WaitingForUA*	No execution ebn.processValidationKey.FM1		The ebn does not send the A661_STRING_CONFIRMED(val) event	The pilot command is not sent to the aircraft system	Loss of control
	Erroneous execution ebn.processValidationKey.FM2		The ebn sends the A661_STRING_CONFIRMED(val) event with a wrong value // sends a wrong A661_Event(val)	A wrong command is sent to the aircraft system	Erroneous control
	Unexpected execution ebn.processValidationKey.FM3		The ebn sends an A661_STRING_CONFIRMED(val) without any user action	A command is sent t the aircraft system without any user action	Erroneous control
ebn.setEntryValidation If the value is true, validate the editing value and change mode to *Idle* ; else, change mode to *Idle* with old value	No execution ebn.setEntryValidation.FM1		The editing value is not validated and is then replaced by the old value	The old value is displayed while the UA is using the new value	Erroneous data display
	Erroneous execution ebn.setEntryValidation.FM2		The editing value is validated (resp. not validated) when it shouldn't (resp. should) have been	The wrong value is displayed while the UA is using another one	Erroneous data display
	Unexpected execution ebn.setEntryValidation.FM3		The editing value is validated (or not) without UA notification	The wrong value is displayed while the UA is using another one	Erroneous data display
ebn.setStyleSet Update the StyleSet parameter with the desired value and send an update notification to the server	No execution ebn.setStyleSet.FM1		The widget does not update its StyleSet parameter	The widget does not modify its display when it should have (the value color is not modified)	Loss of data display
	Erroneous execution ebn.setStyleSet.FM2		The widget updates its StyleSet parameter with a wrong value	The widget does not modify its display when it should have (the value color is modified in a wrong way)	Erroneous data display
	Unexpected execution ebn.setStyleSet.FM3		The widget updates its StyleSet parameter without receiving an setStyleSet request	The widget modifies its display when it should not have (the value color is modified without any reason)	Erroneous data display
ebn.setHighlighted Update the Highlited parameter with the desired value and send an update notification to the server	No execution ebn.setHighlighted.FM1		The widget does not update its Highlighted parameter	The widget stay highlighted (resp. not highlighted) when it should have been not highlighted (reps. highlighted)	Loss of data display
	Erroneous execution ebn.setHighlighted.FM2		The widget updates its Highlighted parameter with a wrong value	The widget stay highlighted (resp. not highlighted) when it should have been not highlighted (reps. highlighted)	Erroneous data display
	Unexpected execution ebn.setHighlighted.FM3		The widget updates its Highlighted parameter without receiving an setHighlighted request	The widget change to highlighted (resp. not highlighted) when it should have stayed not highlighted (resp. highlighted)	Erroneous data display

Tableau 10.3. AMDEC résumant les défaillances de l'*EditBoxNumeric* intervenant dans notre scénario

Identification des modes de défaillance pour le serveur

Nous avons également présenté dans le Chapitre 6 (Tableau 6.4) une AMDEC analysant les modes de défaillances génériques pouvant affecter les fonctions du serveur. L'analyse des modes de défaillance des fonctions du serveur utilisées dans le scénario que nous étudions nous amène à affiner l'AMDEC présentée dans le Chapitre 6. Ainsi, le Tableau 10.4 présente l'AMDEC détaillée des composants du serveur : le `KidsServer` et le `SceneGraph`. Les trois premiers items concernent les événements déclenchés pas les actions utilisateurs (type `inputProcessing` dans le Tableau 6.4) ; les deux items suivants concernent le mode *Caging* offert par le serveur (voir section 9.2.3.1) ; l'item suivant concerne l'identification du widget ciblé par l'action de l'utilisateur ; enfin, le dernier item concerne la mise à jour de la hiérarchie des widgets (autrement dit, du `SceneGraph`).

1	2	3	4	5	6
Item	Failures modes		Local effects	Upper-level effects	Consequence classification
KidsServer.mouseMove Update the picked widget, call setHighlighted function on it and forward the mouseMove event to it	No execution KidsServer.mouseMove.FM1		Upon the receipt of a mouseMove event, the targeted widget is not highlighted / the mouseMove event is not forwarded	The targeted widget is not highlighted	Loss of data display
	Erroneous execution KidsServer.mouseMove.FM2		Upon the receipt of a mouseMove event, a wrong widget is highlighted / a wrong kccu event is forwarded	A wrong widget is highlighted / a wrong control is sent to the aircraft system	Erroneous data display / Erroneous control
	Unexpected execution KidsServer.mouseMove.FM3		The server processes a mouseMove event to without receiving it	A widget is highlighted without any user action	Erroneous data display
KidsServer.mouseClicked Forward the mouseMove event to the picked widget	No execution KidsServer.mouseClicked.FM1		Upon the receipt of a mouseClicked event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution KidsServer.mouseClicked.FM2		Upon the receipt of a mouseClicked event, the server forwards a wrong kccu input event	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution KidsServer.mouseClicked.FM3		The server processes a mouseClicked event to without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
KidsServer.pressNormalKey Forward the pressNormalKey event to the picked widget	No execution KidsServer.pressNormalKey.FM1		Upon the receipt of a pressNormalKey event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution KidsServer.pressNormalKey.FM2		Upon the receipt of a pressNormalKey event, the server forwards a wrong kccu input event	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution KidsServer.pressNormalKey.FM3		The server processes a pressNormalKey event to without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
KidsServer.pressValidationKey Forward the pressValidationKey event to the picked widget	No execution KidsServer.pressValidationKey.FM1		Upon the receipt of a pressValidationKey event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution KidsServer.pressValidationKey.FM2		Upon the receipt of a pressValidationKey event, the server forwards a wrong kccu input event	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution KidsServer.pressValidationKey.FM3		The server processes a pressValidationKey event to without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
KidsServer.openCaging Disable mouse interactions and forward all the keyboard events to the widget that asked for Caging mode	No execution KidsServer.openCaging.FM1		Upon the receipt of an openCaging notification, the server does not activate the Caging mode	The pilot can't interact with the desired widget	Loss of control
	Erroneous execution KidsServer.openCaging.FM2		Upon the receipt of an openCaging notification, the server activate the Caging mode for a wrong widget	The pilot can't interact with the desired widget	Erroneous control
	Unexpected execution KidsServer.openCaging.FM3		The server activate the Caging mode without receiving any notification	The pilot can only interact with the a unique widget without asking for it	Erroneous control
KidsServer.closeCaging Re-enable mouse interactions	No execution KidsServer.closeCaging.FM1		Upon the receipt of a closeCaging notification, the server does not de-activate the Caging mode	The pilot can only interact with the a unique widget when asking for interacting again with all the application	Loss of control
	Erroneous execution KidsServer.closeCaging.FM2		Upon the receipt of a closeCaging notification, the server does not de-activate the Caging mode in a correct way	The pilot can only interact with the a unique widget when asking for interacting again with all the application	Erroneous control
	Unexpected execution KidsServer.closeCaging.FM3		The server de-activate the Caging mode without receiving any notification	The pilot can't interact anymore with the desired widget	Erroneous control
SceneGraph.findWidgetAt Identify the targeted widget	No execution SceneGraph.findWidgetAt.FM1		Upon the receipt of a kccu input event, the server does not forward it	The pilot control is not sent to the aircraft system	Loss of control
	Erroneous execution SceneGraph.findWidgetAt.FM2		Upon the receipt of a kccu input event, the server forwards it to a wrong widget	A wrong control is sent to the aircraft system	Erroneous control
	Unexpected execution SceneGraph.findWidgetAt.FM3		The server sends a kccu input event to a widget without receiving it	A control is sent to the aircraft system without any user action	Erroneous control
SceneGraph.update Update the SceneGraph	No execution SceneGraph.update.FM1		The widget does not update its corresponding parameter	The pilot is not notified of the aircraft system state change	Loss of data display
	Erroneous execution SceneGraph.update.FM2		The widget updates its corresponding parameter with a wrong value	The pilot is notified of a wrong aircraft system state change	Erroneous data display
	Unexpected execution SceneGraph.update.FM3		The widget updates a parameter without receiving an A661_setParameter	The pilot is notified of an aircraft system state change that did not occur	Erroneous data display

Tableau 10.4. AMDEC résumant les défaillances des composants du `Server` (`KidsServer` et `SceneGraph`)

10.3 Identification et formalisation des assertions et de leurs contrôleurs

Comme nous l'avons présenté dans le Chapitre 8, les widgets que nous utilisons dans notre scénario sont des widgets critiques (voir section 8.4.3). Ceci se traduit par le fait que les commandes déclenchées par les actions utilisateurs sur ces widgets ainsi les modifications d'affichage de ceux-ci sont critiques.

Dans le scénario que nous étudions, les contrôles critiques sont déclenchés par la réception par l'UA des événements suivants :

- `A661_STRING_CONFIRMED` provenant de l'*EditBoxNumeric* `SpeedEBN` qui permet de récupérer la nouvelle valeur de vitesse saisie par le pilote.
- `A661_EVT_SELECTION` provenant du *PicturePushButton* `SpeedSelectedPPB` qui permet d'engager la nouvelle vitesse de l'avion.

Les modifications d'affichages critiques sont celles correspondantes à la notification de l'engagement de la nouvelle vitesse par l'autopilote. Elles se résument à la mise à jour du paramètre `StyleSet` de l'*EditBoxNumeric* `SpeedEBN`.

La mise en œuvre de notre architecture logicielle tolérante aux fautes vise donc à détecter les défaillances des fonctions pouvant mener aux pertes de contrôle ou aux contrôles erronés ainsi qu'aux pertes d'affichage et aux affichages erronés lorsque l'on considère les contrôles et les affichages critiques que nous venons de présenter.

Les trois AMDEC que nous avons présentées dans la section précédente nous permettent d'identifier les items et les modes de défaillances à contrôler pour éviter les pertes ou les erronées de contrôles et d'affichage. Ainsi, les pertes de contrôles ou les contrôles erronés peuvent être provoqués par les modes de défaillances des items de type `inputProcessing`, à la fois pour les widgets et pour le serveur. Elles peuvent également être provoquées par une défaillance de la fonction d'identification du widget ciblé par l'action de l'utilisateur. Les pertes d'affichage ou les affichages erronés peuvent être provoqués par les items de type `setParameter` sur les widgets et sur la mise à jour de la hiérarchie des widgets sur le serveur.

Cette identification nous permet de définir les assertions que nous allons contrôler afin de détecter les erreurs (les défaillances des fonctions) pouvant mener aux défaillances du système que nous avons identifiées (pertes ou erronées de contrôle et d'affichages) et qui nous intéressent dans ce scénario. Ces assertions peuvent être divisées en deux types :

- *Les assertions sur les fonctions des widg*ets permettent de détecter les erreurs dans le comportement des widgets.
- *Les assertions sur les fonctions du serveur* permettent de détecter les erreurs dans le comportement du serveur.

Assertions et leurs contrôleurs pour garantir le comportement des widgets

Comme nous l'avons vu dans la section précédente, les widgets sont majoritairement constitués de deux types d'items :

- Les `inputProcessing`, responsables de la gestion des événements résultant des actions de l'utilisateur.
- Les `setParameterProcessing`, responsables de la gestion des demandes de modification de paramètres.

Chacun des items possède trois modes de défaillance (voir Tableau 10.2 et Tableau 10.3) :

- FM1 correspond à la non-exécution de la fonction.
- FM2 correspond à l'exécution erronée de la fonction.
- FM3 correspond à l'exécution autonome de la fonction.

Le Tableau 10.5 rappelle les différents items utilisés dans notre scénario pour les deux widgets (`SpeedEBN` et `SpeedSelectedPPB`).

Widget / Type d'item	EditBoxNumeric SpeedEBN	PicturePushButton SpeedSeletedPPB
inputProcessing	processMouseClicked processNormalKey processValidationKey	processMouseClicked
setParameterProcessing	setStyleSet setHighlighted	setHighlighted
specificItem	setEntryValidation	

Tableau 10.5. Les différents items utilisés dans notre scénario pour les widgets

Chacun de ces items est associé à une assertion décrivant leur bonne exécution. Ainsi, nous pouvons définir de manière générique les deux assertions suivantes, chacune étant vérifiée par deux contrôleurs d'assertions :

- *Pour les items de* type `inputProcessing` : un événement A661_EVENT est envoyé par le widget si et seulement si celui-ci a reçu une demande d'exécution de l'`inputProcessing` correspondant.
- *Pour les items de type* `setParameterProcessing` : la modification du paramètre d'un widget est effectuée si et seulement si celui-ci a reçu une demande d'exécution du `setParameterProcessing` correspondant.

Afin d'illustrer ceci, nous prenons l'exemple d'un item de type `inputProcessing` et d'un item de type `setParameterProcessing` utilisés dans notre scénario (voir Tableau 10.5). Plus particulièrement, nous choisissons les items suivants :

- `processMouseClicked` du *PicturePushButton* `SpeedSelectedPPB`. L'assertion correspondant à l'exécution correcte de cet item est définie dans le Tableau 10.6. Elle est générique à tous les *PicturePushButton*.
- `setStyleSet` de l'*EditBoxNumeric* `SpeedEBN`. L'assertion correspondante à l'exécution correcte de cet item est définie dans le Tableau 10.7. Elle est générique à toutes les *EditBoxNumeric*.

Assertion Definition	*Id*	A1
	item	ppb.processMouseClicked
	Textual definition	If a PicturePushButton receives, while it is enabled and visible, a processMouseMoveClicked function call it must send an A661_EVT_SELECTION
	Formal definition	Let *w* be a PicturePushButton, let *f* = {source, target, functionName, parameters} be a function call, let *We* = {source, eventName, parameters} be a widget event $f = \{source, w, processMouseClicked, parameters\} \wedge w.visible = true \wedge w.enabled = true \Leftrightarrow We = \{w, A661\_EVT\_SELECTION, \emptyset\}$

Tableau 10.6. Tableau définissant l'assertion décrivant l'exécution correcte de l'item processMouseClicked d'un PicturePushButton

Assertion Definition	*Id*	A2
	item	ebn.setStyleSet
	Textual definition	If an EditBoxNumeric receives a setStyleSet function call, it must modify its StyleSet parameter in consequence
	Formal definition	Let *w* be an EditBoxNumeric, let *f* = {source, target, functionName, parameters} be a function call $f = \{source, w, setStyleSet, newValue\} \Leftrightarrow w.StyleSet = newValue$

Tableau 10.7. Tableau définissant l'assertion décrivant l'exécution correcte de l'item setStyleSet d'une EditBoxNumeric

Chacune de ces assertions est associée aux deux contrôleurs d'assertions suivants :

- *Pour les items de* type `inputProcessing` :
    - Le premier contrôleur d'assertion vérifie que si le widget reçoit une demande d'exécution d'un `inputProcessing`, il envoie l'événement `A661_EVENT` correspondant. Ce contrôle permet de détecter les erreurs liées aux modes de défaillance FM1 et FM2. Il est apériodique et déclenché sur la réception d'une demande d'exécution d'un `inputProcessing`.
    - Le second contrôleur d'assertion vérifie que si le widget envoie un événement `A661_EVENT`, il a bien reçu précédemment une demande d'exécution de l'inputProcessing correspondant. Ce contrôle permet de détecter les erreurs liées au mode de défaillance FM3. Il est apériodique et déclenché sur la réception d'un événement `A661_EVENT`.
- *Pour les items de type* `setParameterProcessing` :
    - Le premier contrôleur d'assertion vérifie que si le widget reçoit une demande d'exécution d'un `setParameterProcessing`, il modifie en conséquence la valeur du paramètre correspondant. Ce contrôle permet de détecter les erreurs liées aux modes de défaillance FM1 et FM2. Il est apériodique et déclenché sur la réception d'une demande d'exécution d'un `setParameterProcessing`.
    - Le second contrôleur d'assertion vérifie que si le widget modifie un paramètre, il a bien reçu précédemment une demande d'exécution du `setParameterProcessing` correspondant. Ce contrôle permet de détecter les erreurs liées au mode de défaillance FM3. Il est périodique.

Ainsi, si nous reprenons les exemples ci-dessus, le Tableau 10.8 définit les deux contrôleurs d'assertion associés à l'assertion A1 (voir Tableau 10.6) et le Tableau 10.9 définit les deux contrôleurs d'assertion associés à l'assertion A2 (voir Tableau 10.7).

Assertion Monitor	*Id*	A1.AM1
	Failure mode	ppb.processMouseClicked.FM1 & ppb.processMouseClicked.FM2
	Textual definition	If a PicturePushButton w receives, while it is enabled and visible, a processMouseClicked function call, it send and A661_EVT_SELECTION
	Observables	-
	State image	*w* state
	Frequency	Aperiodic
	Trigger	Reception of a processMouseClicked function call
Assertion Monitor	*Id*	A1.AM2
	Failure mode	ppb.processMouseClicked.FM3
	Textual definition	If a PicturePushButton w sends an A661_EVT_SELECTION, it has received a processMouseClicked function call and is visible and enabled
	Observables	-
	State image	*w* state
	Frequency	Aperiodic
	Trigger	Reception of an A661_EVT_SELECTION event

Tableau 10.8. Tableau définissant les contrôleurs de l'assertion décrivant l'exécution correcte de l'item processMouseClicked d'un PicturePushButton

Assertion Monitor	*Id*	A2.AM1
	Failure mode	ebn.setStyleSet.FM1 & ebn.setStyleSet.FM2
	Textual definition	If an EditBoxNumeric receives a setStyleSet function call, it modifies its styleSet parameter value
	Observables	-
	State image	*w* state
	Frequency	Aperiodic
	Trigger	Reception of a setStyleSet function call
Assertion Monitor	*Id*	A2.AM2
	Failure mode	ebn.setStyleSet.FM3
	Textual definition	If the styleSet parameter value of an EditBoxNumeric is modified, the widget has received a setStyleSet function call
	Observables	w.styleSet
	State image	*w* state
	Frequency	Periodic
	Trigger	-

Tableau 10.9. Tableau définissant les contrôleurs de l'assertion décrivant l'exécution correcte de l'item setStyleSet d'une EditBoxNumeric

Assertions et leurs contrôleurs pour garantir le comportement du serveur

De la même manière que pour les widgets, l'AMDEC que nous avons effectuée sur le serveur nous permet de définir les assertions du serveur.

Les modes de défaillance critiques que nous avons identifiées nous permettent de définir les items critiques des composants du serveur. Tous les items de type `inputProcessing` sont des items critiques car leur défaillance se propagerait et provoquerait une perte ou une erronée de contrôle. Ainsi, par exemple, la non-exécution de l'item `KidsServer.processMouseClicked` impliquerait que l'événement `mouseClicked` ne soit pas transmis au widget ciblé. Dans le cas où le widget ciblé est l'un de nos deux widgets critiques, cela revient au fait qu'il ne reçoit aucune demande d'exécution de l'item `processMouseClicked`, il n'envoie donc par conséquent pas l'événement `A661_EVENT` associé et le contrôle ne sera pas déclenché, ce qui provoque donc une perte de contrôle. L'item `SceneGraph.findWidgetAt` est également critique car c'est lui qui permet de faire l'identification du widget ciblé par l'action de l'utilisateur ; une défaillance de cet item peut donc provoquer une perte ou une erronée de contrôle. Enfin, la mise à jour de la hiérarchie des widgets, effectuée par l'item `SceneGraph.update` est également critique car sa défaillance pourrait provoquer et une perte ou une erronée d'affichage. Ainsi, par exemple, si la visibilité d'un widget n'est pas mise à jour correctement et reste à `false` au lieu de passer à `true`, celui-ci ne sera pas affiché (perte d'affichage), ce qui implique notamment que le pilote ne pourra pas interagir avec ce widget.

Étant donné que nous avons déjà présenté dans le Chapitre 6 un exemple d'assertion et de contrôleur d'assertion associé permettant la vérification du widget identifié par le serveur, nous ne présenterons pas de nouvel exemple d'assertion et de contrôleur d'assertion associé pour le serveur.

10.4 Mise en œuvre de l'architecture autotestable

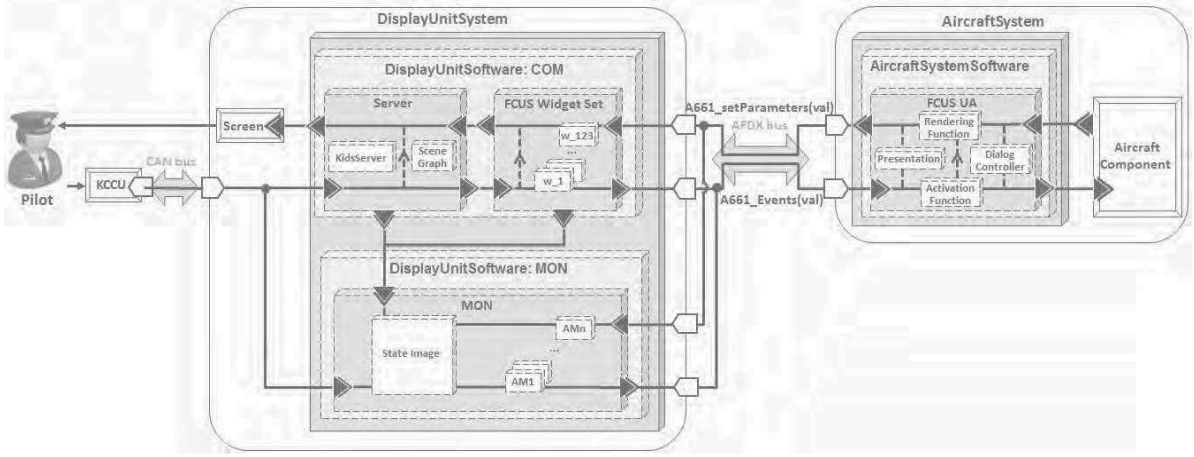


Figure 10.4. Architecture logicielle de l'étude de cas pour la détection des erreurs

Nous avons présenté dans le Chapitre 6 notre architecture logicielle pour la détection des erreurs (voir section 6.3.2). La Figure 10.4 présente l'instanciation de cette architecture logicielle à notre étude de cas. Nous retrouvons sur cette architecture les composants logiciels du système interactif n'intégrant pas la détection des erreurs (voir section 10.1) : le composant `COM` du CDS (constitué du serveur et de l'ensemble des widgets de l'application) et l'UA (constituée de la présentation, des fonctions d'activation et de rendu et du contrôleur de dialogue). Pour permettre la détection des erreurs dans le CDS, nous avons ajouté à ces composants le composant `MON` du CDS, constitué d'une *image de l'état* du composant `COM` (`State Image`) ainsi que des différents contrôleurs d'assertions (`AM1` … `AMn`).

Architecture de la maquette

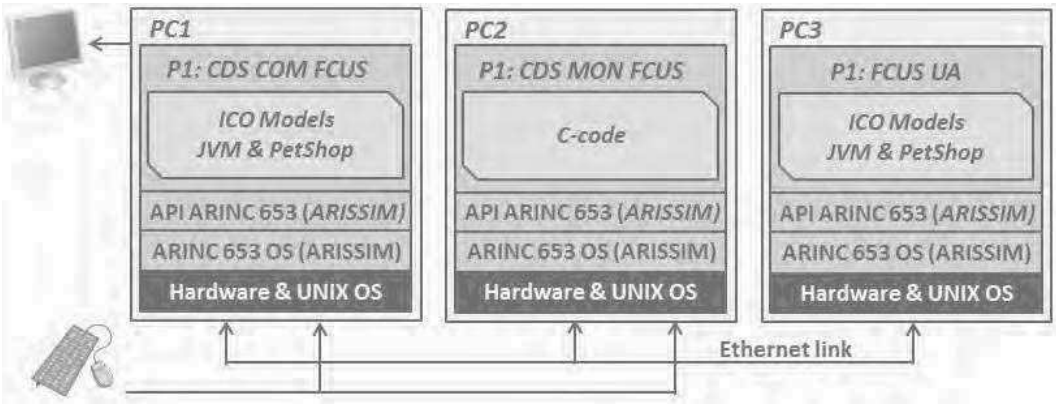


Figure 10.5. Architecture logicielle et matérielle de notre maquette

La Figure 10.5 présente l'architecture logicielle et matérielle de la maquette, que nous avons présentée dans le Chapitre 7, instanciée à notre étude de cas. Celle-ci est développée sur un réseau de trois ordinateurs. Chacun de ces ordinateurs est équipé d'un système d'exploitation Linux et exécute le simulateur ARISSIM (voir section 7.2.3). Le premier ordinateur (`PC1`) exécute le code correspondant au composant `COM` du CDS dans une partition `P1`. Le deuxième ordinateur (`PC2`) exécute le code correspondant au composant `MON` du CDS dans une partition `P1`. Enfin, le troisième ordinateur (`PC3`) exécute le code correspondant à l'UA associé dans une partition `P1`.

Implémentation du composant COM

Le composant `COM` du CDS est développé en utilisant notre approche à base de modèles (voir Chapitre 5). Ainsi, il est développé de la même manière que nous l'avons présenté dans le Chapitre 9 : ses différents composants logiciels sont modélisés à l'aide du formalisme ICO. Son architecture logicielle est présentée en Figure 10.6. Cette figure met en évidence les différents composants logiciels

du composant COM : les composants du serveur (KidsServer et SceneGraph) ainsi que les différents widgets (w_1 … w_123). Les modèles ICO de ces différents composants ont été présentés dans le Chapitre 9. Ces différents composants sont exécutés par l'outil PetShop (voir section 7.1) qui s'exécute sur une machine virtuelle Java dans la partition P1 du PC1.

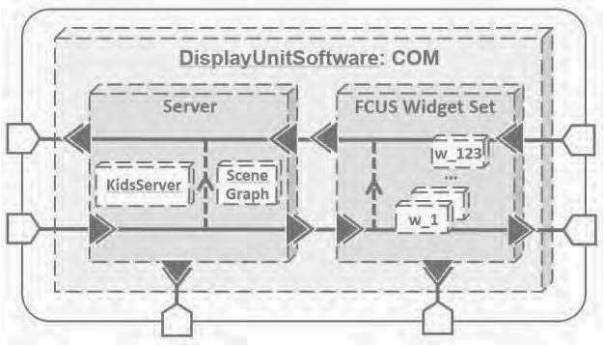


Figure 10.6. Architecture logicielle du composant COM

En plus des modèles ICO des différents composants, le composant COM implémente un objet Java permettant de gérer la communication entre le simulateur et les modèles afin de :

- *Gérer les communications entre le CDS et l'UA*.

  Il est abonné aux différents événements A661_EVENT des widgets à destination de l'UA et les transmet à travers l'utilisation d'un canal de communication en mode queuing fourni par le simulateur ARISSIM. Il réceptionne également grâce à l'utilisation d'un canal de communication en mode queuing les appels de fonctions provenant de l'UA et les transmet aux différents widgets. Enfin, il réceptionne les événements provenant des actions de l'utilisateur et les transmet au serveur.

- *Gérer les communications entre le COM et le MON.*

  Il écoute, à travers les fonctionnalités de l'interprète des modèles ICO, les différents observables nécessaires au MON pour exécuter ses contrôleurs d'assertions. Il transfère ses informations à travers l'utilisation de deux canaux de communication, le premier en mode queuing et le second en mode sampling. Le canal de communication en mode queuing est utilisé pour transmettre les événements (tels que les événements envoyés par les widgets à l'UA) et le canal de communication en mode sampling est utilisé pour transmettre des valeurs de variables (tels que les paramètres constituant l'état d'un widget).

Implémentation du composant MON

Afin de le diversifier au maximum vis-à-vis du composant COM et de pouvoir détecter les fautes qui peuvent affecter le support d'exécution des modèles ICO (l'outil PetShop et la machine virtuelle Java), le composant MON est implanté en langage C.

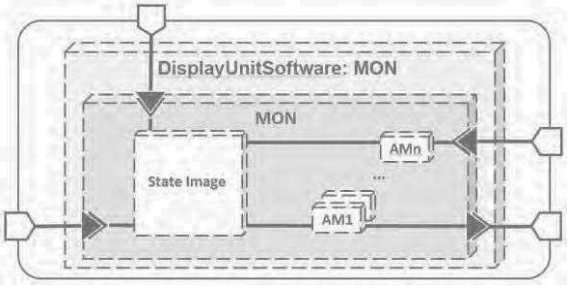


Figure 10.7. Architecture logicielle du composant MON

Comme le montre la Figure 10.7 qui présente son architecture logicielle, il est constitué d'une *image de l'état* du composant COM (State Image) ainsi que des différents contrôleurs d'assertions (AM1 … AMn). Chacun de ces composants logiciels correspond à une fonction codée en langage C. Le programme principal du composant MON est responsable de la communication avec le simulateur. Il

est responsable de la mise à jour l'image de l'état en fonction des événements qu'il réceptionne. Cette image est principalement constituée des différents widgets de l'application, stockés sous la forme d'un tableau dans lequel chaque ligne correspond à un widget et à ses différents paramètres. Le programme principal est également responsable de la gestion du déclenchement des différents contrôleurs d'assertions en fonction des événements qu'il réceptionne ou de leur périodicité. Il est exécuté sur la partition P1 du PC2.

Afin d'illustrer l'implémentation d'un contrôleur d'assertion en langage C, la Figure 10.8 présente l'implémentation en langage C des deux contrôleurs de l'assertion A1 permettant la détection des erreurs affectant l'item `processMouseClicked` d'un *PicturePushButton*. De la même manière, la Figure 10.9 présente l'implémentation en langage C des deux contrôleurs de l'assertion A2 permettant la détection des erreurs affectant l'item `setSyleSet` d'une *EditBoxNumeric*.

A1.AM1: ppb.processMouseClicked.assert

```
//MON state
boolean w.visible, w.enabled;

// ppb.processMouseClicked.assert
int errorDetected = -1;

if (functionCall == {source, w, processMouseClicked, parameters}){
    if (w.visible == true && w.enabled == true){
            boolean timeOut = startTimer();
    }
}
while (!timeOut){
    if (! timeOut && widgetEvent.contains({w,A661_EVT_SELECTION,Ø}){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
    }
}
if (timeOut && errorDetected == -1){
    errorDetected = 1;
    sendError(functionCall, errorDetected);
}
```

Ce contrôleur d'assertion est déclenché par la réception d'une demande d'exécution de la fonction processMouseCliked d'un PicturePushButton. Dans ce cas, un timer est déclenché. Si aucun événement A661_EVT_SELECTION n'est envoyé par le PicturePushButton dans le temps imparti, une erreur est envoyée à l'UA.

A1.AM2: A661_EVT_SELECTION.assert

```
//MON state
boolean w.visible, w.enabled;


// ppb.processMouseClicked.assert
int errorDetected = 0;

if (widgetEvent == {w,A661_EVT_SELECTION, Ø}){
    if (functionCall.contains({source, w, processMouseClicked, parameters}) && w.visible ==
true                    && w.enabled == true){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
    }else{
            errorDetected = 1;
            sendError(functionCall, errorDetected);
    }
}
```

Ce contrôleur d'assertion est déclenché par la reception d'un événement A661_EVT_SELECTION. Dans ce cas, si le PicturePushButton ayant envoyé cet événement n'a pas reçu de demande d'exécution de la fonction processMouseCliked, une erreur est envoyée à l'UA.

Figure 10.8. Implémentation en C des contrôleurs d'assertions A1.AM1 et A1. AM2

A2.AM1: ebn.setStyleSet.assert

```
//MON state
int w.styleSet;

//Observables
int w.styleSetObserved

//ebn.setStyleSet.assert
int errorDetected = -1;

if (functionCall == {source, w, setStyleSet, newValue}){
    boolean timeOut = startTimer();
}
while (!timeOut){
    if (! timeOut && w.styleSetObserved == newValue){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
    }
}
if (timeOut && errorDetected == -1 && w.styleSetObserved != newValue){
    errorDetected = 1;
    sendError(functionCall, errorDetected);
}
```

Ce contrôleur d'assertion est déclenché par la réception d'une demande d'exécution de la fonction setStyleSet d'une EditBoxNumeric. Dans ce cas, si la valeur obervées du paramètre styleSet de l'EditBoxNumeric n'a pas été mise à jour correctement, une erreur est envoyée à l'UA.

A1.AM2: ebn.styleSetParameterValue.assert

```
//MON state
int w.styleSet;

//Observables
int w.styleSetObserved

//ebn.styleSetParameterValue.assert
int errorDetected = 0;

if (w.styleSet != w.styleSetObserved){
    if functionCall.contains({source, w, setStyleSet, w.styleSet}){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
    }else{
            errorDetected = 1;
            sendError(functionCall, errorDetected);
    }
}
```

Ce contrôleur d'assertion est périodique. À chacun de ces déclenchements, il compare la valeur observée du paramètre styleSet de l'EditBowNumeric avec celle qu'il conserve dans son image de l'état. Si elles sont différentes et que l'EditBoxNumeric n'a pas reçu de demande d'exécution de la fonction setStyleSet avec la valeur correspondante à la valeur obervée du paramètre, une erreur est envoyée à l'UA.

Figure 10.9. Implémentation en C des contrôleurs d'assertions A2.AM1 et A2.AM2

Implémentation de l'UA

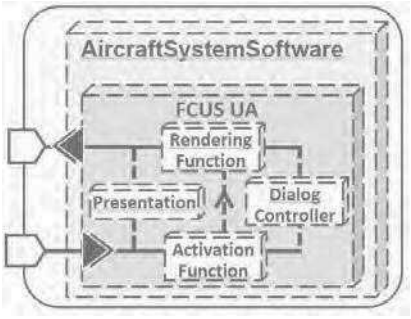


Figure 10.10. Architecture logicielle de l'UA

De la même manière que le composant `COM`, l'UA est développée en utilisant notre approche à base de modèles. La Figure 10.10 présente l'architecture logicielle de l'UA et nous permet de mettre en évidence ces différents composants logiciels : `Presentation`, `Activation Function`, `Rendering Function` et `Dialog Controller`. Ils sont modélisés à l'aide de la notation ICO et sont exécutés par l'outil PetShop qui s'exécute sur une machine virtuelle Java dans la partition `P1` du `PC3`.

En plus des modèles ICO des différents composants, l'UA implémente également un objet Java permettant de gérer la communication entre le simulateur et les modèles afin de :

- *Gérer les communications entre l'UA et le COM.*

  Il réceptionne, à travers un canal de communication en mode queuing, les différents événements A661_EVENT envoyés par le composant MON et les transmet sous forme d'événements ICO aux modèles. Il permet également d'envoyer, toujours au travers de l'utilisation d'un canal de communication en mode queuing, les appels de méthodes `A661_setParameter` qui sont effectués par l'UA.

- *Gérer les communications entre l'UA et le MON.*

  Il réceptionne, à travers un canal de communication en mode queuing, les différents événements de notification d'erreur provenant du composant `MON` et les transmet aux modèles ICO sous forme d'événements ICO. Il permet également de transmettre au composant MON les différents appels de méthodes `A661_setParameter` qui sont effectués par l'UA.

Les modèles ICO de ces différents composants sont les mêmes que ceux qui ont été présentés dans le Chapitre 9 à une exception près : ils embarquent des fonctionnalités supplémentaires pour le traitement des erreurs. Pour cette étude de cas, nous avons choisi de notifier le pilote lorsqu'une erreur est détectée. Celui-ci peut alors décider, par exemple, d'utiliser un autre système ou de reconfigurer l'application FCUS sur une autre DU. Cette notification est faite par l'affichage d'un Label spécial comme présenté en Figure 10.11.

Le modèle ICO de la gestion des erreurs dans le contrôleur de dialogue est présenté en Figure 10.12. Ainsi, le système possède deux états : un état sans erreur représenté par la présence d'un jeton dans la place NoError pendant lequel le système se comporte normalement et un état avec erreur représenté par la présente d'un jeton dans la place Error pendant lequel le message d'erreur est affiché. Le passage de l'état sans erreur à l'état avec erreur est déclenché par la réception d'un événement d'erreur. Le passage de l'état avec erreur à l'état sans erreur peut-être déclenché par la réception d'un événement notifiant le recouvrement de l'erreur.

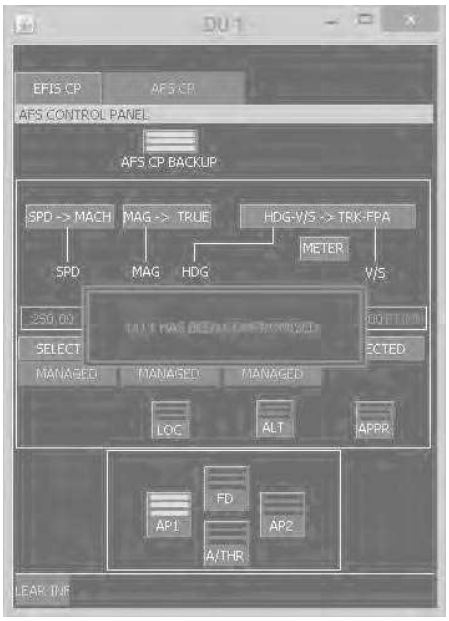


Figure 10.11. Notification d'erreur pour le pilote

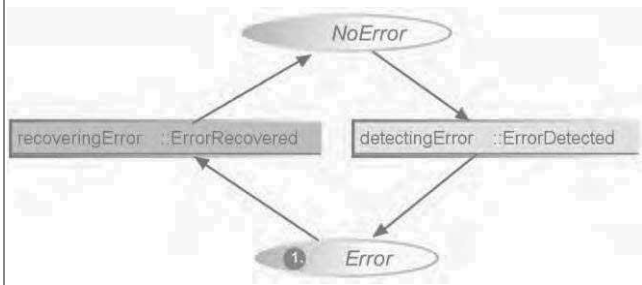


Figure 10.12. Modèle ICO de la gestion des erreurs

10.5 Synthèse

Dans ce chapitre, nous avons présenté la mise en œuvre notre architecture logicielle pour la détection des erreurs (présentée dans le Chapitre 6) sur l'étude de cas que nous avons présentée dans le Chapitre 8 : l'application interactive FCUS.

Ainsi, nous avons tout d'abord présenté l'analyse de notre étude de cas au travers de l'étude d'un scénario d'utilisation d'une des fonctions critiques de l'application. En nous appuyant sur ce scénario, nous avons déterminé les modes de défaillance des différentes fonctions par la réalisation d'AMDEC. Ces AMDEC nous permettent de définir les assertions décrivant le bon fonctionnement du système ainsi que les contrôleurs d'assertion permettant la détection des erreurs.

Nous avons ensuite présenté l'architecture logicielle finale de notre étude de cas. Cette architecture permet la détection des erreurs dans les composants génériques (serveur et widgets) de notre étude de cas, ce qui permet de notifier le pilote en cas d'erreur.

Enfin, nous avons présenté la maquette que nous avons mise en œuvre pour valider la faisabilité de notre architecture.

Cette implémentation a un but de preuve de concept. En effet, l'environnement pratique de la thèse ne permet pas d'obtenir des résultats réalistes (ce qui n'était pas dans le périmètre de la thèse). Une implémentation plus réaliste sur un vrai noyau ARINC 653 permettrait des mesures plus réalistes ainsi que d'effectuer des optimisations, des mesures de performances ainsi que l'influence de notre architecture par rapport aux problèmes de synchronisation entre les différents composants, à la perception de l'utilisateur, à d'éventuels temps de retard, etc…

CONCLUSION ET PERSPECTIVES

L'utilisation de systèmes interactifs pour la commande et le contrôle de systèmes avioniques dans les cockpits d'avions civils apporte de nombreux avantages tels que la réduction de la charge de travail du pilote et l'augmentation de l'évolutivité du cockpit. Ces systèmes ont été introduits dans les cockpits d'avions civils au début des années 2000. Cependant, pour des raisons d'exigences de sûreté de fonctionnement, l'utilisation de ces systèmes est limitée à l'heure actuelle à la commande et au contrôle de fonctions non critiques. L'objectif de nos travaux était de démontrer qu'il est possible de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes avioniques critiques.

Contributions

La contribution principale de cette thèse consiste à proposer une approche permettant de concevoir et développer des systèmes interactifs avec un niveau de sûreté de fonctionnement et d'utilisabilité suffisant pour qu'ils puissent être utilisés pour la commande et le contrôle de systèmes critiques.

Pour cela, nous avons tout d'abord défini, dans le Chapitre 4, le périmètre des travaux et les hypothèses que nous avons retenues. Ainsi, nous nous intéressons dans cette thèse à la sûreté de fonctionnement des systèmes interactifs et plus particulièrement à celle de leurs composants logiciels responsables des fonctionnalités interactives. Nous avons ensuite proposé une classification des défaillances des systèmes interactifs et présenté le modèle de faute auquel nous nous intéressons dans cette thèse : les fautes logicielles de développement et les fautes matérielles pouvant affecter le système durant son exécution. Nous avons également proposé, dans ce chapitre, une architecture générique pour les systèmes interactifs. Celle-ci nous a permis d'identifier les composants logiciels sur lesquels nous appliquons notre approche pour garantir la sûreté de fonctionnement des systèmes interactifs.

Notre approche pour concevoir et développer des systèmes interactifs sûrs de fonctionnement est duale, elle repose tout d'abord sur une conception et un développement de systèmes interactifs zéro-défaut et ensuite sur l'introduction des mécanismes de tolérance aux fautes.

Notre approche vers une conception zéro-défaut des systèmes interactifs, présentée dans le Chapitre 5, consiste en une spécification et un développement à base de modèles des différents composants logiciels d'un système interactif au travers l'utilisation d'une technique de description formelle. Nous avons pour cela choisi la notation formelle ICO qui permet de modéliser les différents aspects nécessaires à la description des composants logiciels des systèmes interactifs. Enfin, nous avons montré comment tous les composants logiciels de l'architecture définie dans le Chapitre 4 peuvent être modélisés à l'aide de cette notation formelle.

Notre approche de conception des systèmes interactifs tolérants aux fautes, présentée dans le Chapitre 6, repose sur une architecture logicielle basée sur la notion de mécanisme autotestable. Cette architecture repose sur la définition et la réalisation d'un composant chargé de la vérification de l'exécution des fonctions du système interactif, c'est-à-dire la commande ou COM. Le composant de *monitoring*, appelé MON, est constitué de différents contrôleurs d'assertions. Afin de permettre la réalisation de cette architecture, nous avons proposé une méthode pour la définition des contrôleurs d'assertions s'appuyant sur l'utilisation d'une analyse des modes de défaillance des composants logiciels fonctionnels concernés. Nous nous appuyons sur une analyse des modes de défaillance selon la méthode des AMDEC. Cette analyse s'effectue à l'aide d'une description de l'architecture logicielle du système interactif avec le langage graphique AADL et de diagrammes de séquence représentant le comportement du système.

Nous avons ensuite présenté, dans le Chapitre 7, des moyens de mise en œuvre de notre approche. Ainsi, l'approche zéro-défaut peut être mise en œuvre au travers de l'utilisation d'un outil, nommé

PetShop, permettant l'édition de modèles ICO, leur exécution en temps réel ainsi que leur analyse. L'architecture logicielle tolérante aux fautes impose des propriétés de confinement d'erreur reposant sur du partitionnement spatial et temporel, ce que l'on trouve dans les architectures de systèmes avioniques critique en suivant le standard ARINC 653. Dans nos travaux, nous avons utilisé un simulateur de système d'exploitation avionique, nommé ARISSIM, fournissant l'isolation des différents composants logiciels de l'architecture tolérante aux fautes. Notre objectif ici consistait à réaliser une preuve de concepts. Une implémentation sur un noyau natif avionique est nécessaire aujourd'hui pour passer à l'échelle et effectuer des mesures de performance réalistes.

Enfin, nous avons pu ainsi valider la faisabilité de notre approche par son application à une étude de cas réaliste issue du milieu de l'avionique : un système interactif inspiré du Flight Control Unit. Ainsi, nous avons présenté, dans le Chapitre 9, l'application de notre approche à bases de modèles permettant de concevoir des systèmes interactifs zéro-défaut. Ceci a été effectué en réalisant la modélisation à l'aide de la notation formelle ICO des différents composants logiciels de cette application. Nous avons également présenté, dans le Chapitre 10, l'application de notre architecture logicielle et sa mise en œuvre sur l'étude de cas FCUS. Ceci a été effectué en exemplifiant celle-ci au travers l'étude d'un scénario d'utilisation de l'application nous permettant d'appliquer la méthode proposée pour la définition des contrôleurs d'assertions.

Pour conclure, nous avons présenté une approche duale et homogène de prévention et de tolérance aux fautes pour concevoir et développer des systèmes interactifs tolérants aux fautes. Celle-ci repose à la fois sur une approche de prévention des fautes logicielles de développement et sur une approche de tolérance aux fautes logicielles résiduelles et aux fautes naturelles en opération. Pour développer cette approche, nous avons proposé une solution architecturale fondée sur le principe des architectures autotestables. Enfin, pour montrer la faisabilité de notre solution, nous l'avons appliquée à un exemple grâce à l'utilisation d'un simulateur de noyau ARINC 653. Cette implémentation a un but de preuve de concept. En effet, l'environnement pratique de la thèse ne permet pas d'obtenir des résultats réalistes (ce qui n'était pas dans le périmètre de la thèse). Une implémentation plus réaliste sur un vrai noyau ARINC 653 permettrait des mesures plus réalistes ainsi que d'effectuer des optimisations, des mesures de performances ainsi que l'influence de notre architecture par rapport aux problèmes de synchronisation entre les différents composants, à la perception de l'utilisateur, à d'éventuels temps de retard… Une optimisation de la programmation événementielle du moniteur ainsi qu'au niveau de la transmission des événements sur un prototype réaliste est un objectif d'ingénierie industrielle aujourd'hui et sort du cadre de nos recherches dans cette thèse.

Limites et perspectives

La réalisation d'une implémentation réaliste de notre approche nécessite des analyses complémentaires qui étaient hors du périmètre de la thèse et nécessitent des moyens d'ingénierie industriels conséquents.

Premièrement, nous avons présenté une approche à base de modèles, fondée sur la modélisation du comportement des différents composants logiciels d'un système interactif à l'aide d'une notation formelle. Cependant, l'étude de l'intégration de cette approche dans un processus de développement complet allant de la spécification du système jusqu'à son intégration dans un cockpit d'avion était hors du périmètre de la thèse. Celle-ci nécessite la mise en place de moyens de vérification et de validation du logiciel. Comme nous l'avons présenté dans le Chapitre 7, l'utilisation de la notation formelle ICO présente l'avantage de fournir un support à ces activités. Cependant, l'analyse formelle sur les modèles ICO nécessite encore des extensions telles que la formalisation des mécanismes de communication par événements et la validation de sa faisabilité sur un cas d'étude concret.

Deuxièmement, nous avons présenté une architecture logicielle tolérante aux fautes et nous avons validé sa faisabilité. Cependant, afin de pouvoir utiliser cette architecture dans un système avionique, elle doit être validée en termes de couverture des mécanismes de tolérance. En effet, il est important de garantir que les mécanismes de tolérance aux fautes détectent toutes les erreurs pouvant mener à la

défaillance du système. Ceci peut être réalisé, comme nous l'avons présenté dans le Chapitre 7, en menant une campagne d'injection de fautes permettant de vérifier la couverture de détection du composant chargé de réaliser le contrôle des différentes assertions définies.

Troisièmement, nous avons présenté une implémentation de notre approche sur un simulateur. Cette implémentation a un but de preuve de concept. Une implémentation réelle nécessite d'envisager la manière dont le logiciel sera finalement implanté. En effet, celui-ci peut être implanté comme nous l'avons montré dans notre étude de cas en exécutant les modèles ICO à l'aide d'un interprète de réseau de Petri. Cependant, il est également envisageable d'utiliser les modèles ICO comme simple spécification du logiciel final. Dans ce cas, l'ajout d'une capacité de génération automatique de code à partir des modèles ICO peut être une piste intéressante comme support à l'implantation du logiciel.

Enfin, il est important de garantir que les mécanismes mis en place n'affectent pas l'utilisabilité du système afin de ne pas augmenter considérablement la charge de travail du pilote. Pour cela, nous avons proposé dans la section 7.3 une approche s'appuyant sur les modèles de tâches pour comparer l'utilisabilité de deux systèmes et évaluer ainsi l'impact de l'utilisation de notre approche sur l'utilisabilité du système. Cette approche permet de vérifier que les modifications mises en place pour augmenter le niveau de sûreté de fonctionnement n'impactent pas l'utilisabilité du système. Cependant, cette approche nécessite d'être améliorée par une analyse plus fine des tâches afin de calculer de manière plus fiable la charge de travail du pilote. Il est également essentiel de valider l'applicabilité de cette approche dans un processus de conception et de développement global. De plus, il peut être également intéressant d'étudier différents moyens de recouvrement que celui que nous avons proposé. En effet, certaines des erreurs détectées peuvent être recouvertes de manière automatique sans en informer le pilote, ce qui diminuerait sa charge de travail.

RÉCAPITULATIF DES ARTICLES PUBLIÉS

Articles de journaux internationaux

2015 Célia Martinie, Philippe Palanque, Racim Fahssi, Jean-Paul Blanquart, Camille Fayollas, Christel Seguin. *Task Model-Based Systematic Analysis of Both System Failures and Human Errors*, IEEE Transactions on Human-Machine Systems. À paraître, 12 pages.

2013 Camille Fayollas, Jean-Charles Fabre, Philippe Palanque, Éric Barboni, David Navarre, Yannick Deleris. *Interactive cockpits as critical applications: a model–based and a fault–tolerant approach*. International Journal of Critical Computer-Based Systems, Vol. 4, N°3, pp. 202-226.

Articles de conférences internationales avec comité de lecture et actes

2015 Célia Martinie, David Navarre, Philippe Palanque, Camille Fayollas. *A Generic Tool-Supported Framework for Coupling Task Models and Interactive Applications*. In Proc. of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2015). À paraître, 10 pages.

2014 Camille Fayollas, Jean-Charles Fabre, Philippe Palanque, Martin Cronel, David Navarre, Yannick Deleris. *A Software-Implemented Fault-Tolerance Approach for Control and Display Systems in Avionics*. In Proc. of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2014), IEEE, pp. 21-30.

2014 Camille Fayollas, Célia Martinie, Philippe Palanque, Yannick Deleris, Jean-Charles Fabre, David Navarre. *An Approach for Assessing the Impact of Dependability on Usability: Application to Interactive Cockpits*. In Proc. of the 10th European Dependable Computing Conference (EDCC 2014), IEEE, pp. 198-209.

2011 Adrienne Tankeu-Choitat, David Navarre, Philippe Palanque, Yannick Deleris, Jean-Charles Fabre, Camille Fayollas. *Self-checking components for dependable interactive cockpits using formal description techniques*. In Proc. of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011), IEEE, pp. 164-173.

Articles de conférences internationales francophones avec comité de lecture et actes

2014 Camille Fayollas, Philippe Palanque, Jean-Charles Fabre, David Navarre, Éric Barboni, Martin Cronel, Yannick Deleris. *Résilience des systèmes interactifs : contribution par une architecture tolérante aux fautes*. Dans les Proc. de la 26ème Conférence sur l'Interaction Homme-Machine (IHM 2014), ACM, pp. 80-90

Articles de workshop internationaux avec comité de lecture et actes

2014 Camille Fayollas, Célia Martinie, David Navarre, Philipe Palanque, Racim Fahssi. *Fault-Tolerant User Interfaces for Critical Systems: Duplication, Redundancy and Diversity as New Dimensions of Distributed User Interfaces*. In Proc. of the 4th Workshop on Distributed User Interfaces and Multimodal Interaction (DUI 2014). Proceedings of DUI 2014, 4p.

2013 José-Luis Silva, Camille Fayollas, Arnaud Hamon, Philippe Palanque, Célia Martinie, Éric Barboni. *Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification.* In Proc. of the 5th International Workshop on Formal Methods for Interactive Systems (FMIS 2013), Elsevier, 14p.

2012 Camille Fayollas, Jean-Charles Fabre, David Navarre, Philippe Palanque, Yannick Deleris. *Fault-Tolerant Interactive Cockpits for Critical Applications: Overall Approach*. In Software Engineering for Resilient Systems (SERENE 2012), Springer Berlin Heidelberg, pp. 32-46.

Présentation à des rencontres doctorales

2013 Camille Fayollas. *Addressing dependability for interactive systems: application to interactive cockpits*. In Proc. of the 5th ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS 2013), ACM, pp. 163-166.

Articles de conférences internationales avec actes et acceptation sur résumé

2014 Camille Fayollas, Philippe Palanque, Jean-Charles Fabre, David Navarre, Yannick Deleris, Arnaud Hamon. *A Fault-Tolerant Software Architecture and its Formal Specification for Embedded, Real-Time Interactive Systems*. In Proc. of the International Conference on Embedded Real Time Software and Systems (ERTS^2 2014). Proceedings of ERTS² 2014.

BIBLIOGRAPHIE

AEEC. “ARINC 651.” *ARINC specification 651 - Design Guidance for Integrated Modular Avionics.* 1991.

—. “ARINC 653.” *ARINC specification 653 - Avionics Application Software Standard Interface.* 2003.

—. “ARINC 661.” *ARINC specification 661 - Cockpit Display System Interfaces to User Systems.* 2013.

—. “ARINC 664-P7.” *ARINC specification 664 - P7 - Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network.* 2009.

Appert, Caroline, and Michel Beaudouin-Lafon. “SwingStates: Adding State Machines to the Swing Toolkit.” *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2006. 319-322.

Arlat, J., et al. “Fault injection for dependability validation: a methodology and some applications.” *Software Engineering, IEEE Transactions on* 16, no. 2 (Feb 1990): 166-182.

Arlat, Jean, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie, and David Powell. “Tolérance aux fautes.” In *Encyclopédie de l'informatique et des systèmes d'informations*, 240-270. Paris: Vuibert, 2006.

Avizienis, Algirdas. “The N-Version Approach to Fault-Tolerant Software.” *Software Engineering, IEEE Transactions on* SE-11, no. 12 (Dec 1985): 1491-1501.

Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing.” *IEEE Trans. Dependable Sec. Comput.* 1, no. 1 (2004): 11-33.

Baeten, Jos C. M. “A Brief History of Process Algebra.” *Theor. Comput. Sci.* (Elsevier Science Publishers Ltd.) 335, no. 2-3 (2005): 131-146.

Barboni, Eric. “Méthodes formelles pour les composants logiciels appliquées aux systèmes interactifs critiques.” Ph.D. dissertation, Université Toulouse III - Paul Sabatier, 2006.

Barboni, Eric, Jean-François Ladry, David Navarre, Philippe Palanque, and Marco Winckler. “Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models.” *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2010. 165-174.

Barboni, Eric, Philippe Palanque, David Navarre, and Sandra Basnyat. “A Formal Description Technique for Interactive Cockpit Applications Compliant with ARINC Specification 661.” *Industrial Embedded Systems, SIES '07* . Lisbon: IEEE, 2007.

Barboni, Eric, Rémi Bastide, Xavier Lacaze, David Navarre, and Philippe Palanque. “Petri Net Centered versus User Centered Petri Nets Tools.” *AWPN 2003 - 10th Workshop Algorithms and Tools for Petri Nets, Eichstätt, Germany, 26/09/2003-27/09/2003*. 2003.

Barbosa, Raul, Johan Karlsson, Henrique Madeira, and Marco Vieira. “Fault Injection.” In *Resilience Assessment and Evaluation of Computing Systems*, edited by Katinka Wolter, Alberto Avritzer, Marco Vieira and Aad van Moorsel, 263-281. Springer Berlin Heidelberg, 2012.

Barbosa, Ricardo, Nuno Silva, and JoãoMário Cunha. *csXception®: First Steps to Provide Fault Injection for the Development of Safe Systems in Automotive Industry.* Vol. 7869, in *Dependable Computing*, edited by Marco Vieira and JoãoCarlos Cunha, 202-205. Springer Berlin Heidelberg, 2013.

Basili, Victor R., and Barry T. Perricone. “Software Errors and Complexity: An Empirical Investigation.” *Commun. ACM* (ACM) 27, no. 1 (1984): 42-52.

Bass, Len, et al. “A Metamodel for the Runtime Architecture of an Interactive System.” *SIGCHI Bull.* (ACM) 24, no. 1 (1992): 32-37.

Bastide, Rémi, and Philippe A Palanque. “Petri net objects for the design, validation and prototyping of user-driven interfaces.” *Interact.* 1990. 625-631.

Bastide, Rémi, and Philippe A. Palanque. “A Visual and Formal Glue between Application and Interaction.” *J. Vis. Lang. Comput.* 10, no. 4 (1999): 481-507.

Bastide, Rémi, David Navarre, Philippe Palanque, Amélie Schyn, and Pierre Dragicevic. “A Model-based Approach for Real-time Embedded Multimodal Systems in Military Aircrafts.” *Proceedings of the 6th International Conference on Multimodal Interfaces.* New York, NY, USA: ACM, 2004. 243-250.

Beaudouin-Lafon, Michel. *Ingénierie des Systèmes Interactifs.* 1997. http://www.lri.fr/~mbl/ENS/IHM/ecole-in2p3/Cours/cours1.html (accessed 04 29, 2015).

Beaussart, Lucie, Thomas Bétous, Abdelkader Bouarfa, and William Excoffon. “Développement avancé d'un système embarqué "satellite AGILE" sur carte Raspberry PI.” Rapport de projet long INP-ENSEEIHT, 2014, 31.

Bedoin, Yann, Robin Fraudeau, Jordan Stekke, and Yassine Toure. “Extension d'une maquette de satellite agile et analyse de sûreté de fonctionnement en fonction du niveau d'alimentation.” Rapport de projet long INP-ENSEEIHT, 2015, 36.

Bernhaupt, Regina. “User Experience Evaluation Methods in the Games Development Life Cycle.” In *Game User Experience Evaluation*, edited by Regina Bernhaupt. Springer International Publishing Switzerland 2015, 2015.

Blanch, Renaud, and Michel Beaudouin-Lafon. “Programming Rich Interactions Using the Hierarchical State Machine Toolkit.” *Proceedings of the Working Conference on Advanced Visual Interfaces.* New York, NY, USA: ACM, 2006. 51-58.

Bowen, Judy, and Steve Reeves. “Modelling User Manuals of Modal Medical Devices and Learning from the Experience.” *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems.* New York, NY, USA: ACM, 2012. 121-130.

Brat, Guillaume, Célia Martinie, and Philippe Palanque. *V&V of Lexical, Syntactic and Semantic Properties for Interactive Systems through Model Checking of Formal Description of Dialog.* Vol. 8004, in *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments*, edited by Masaaki Kurosu, 290-299. Springer Berlin Heidelberg, 2013.

Brooke, John. “SUS-A quick and dirty usability scale.” *Usability evaluation in industry* (London) 189, no. 194 (1996): 4-7.

Bustamante, Gabriel, and Rémi Palustran. “Simulation d'un noyau temps-réel avionique ARINC 653 sur Linux.” Rapport de projet long INP-ENSEEIHT, 2013, 44.

Card, Stuart K., Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983.

Cardelli, Luca, and Rob Pike. “Squeak: A Language for Communicating with Mice.” *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 1985. 199-204.

Carr, David A. “Specification of Interface Interaction Objects.” *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* New York, NY, USA: ACM, 1994. 372-378.

Chereque, Marc, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. “Active replication in Delta-4.” *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on.* 1992. 28-37.

Chillarege, Ram, et al. “Orthogonal defect classification-a concept for in-process measurements.” *Software Engineering, IEEE Transactions on* 18, no. 11 (Nov 1992): 943-956.

Coninx, Karin, Erwin Cuppens, Joan De Boeck, and Chris Raymaekers. “Integrating Support for Usability Evaluation into High Level Interaction Descriptions with NiMMiT.” *Proceedings of the 13th International Conference on Interactive Systems: Design, Specification, and Verification*. Berlin, Heidelberg: Springer-Verlag, 2007. 95-108.

Coutaz, Joëlle. “PAC, on Object Oriented Model for Dialog Design.” *Interact'87*. 1987.

Coutaz, Joelle, Fabio Paterno, Giorgio Faconti, and Laurence Nigay. “A comparison of approaches for specifying multi-modal interactive systems.” *Proceedings of the ERCIM Workshop on Multimodal Human-Computer Interaction*. 1993. 165-174.

Cronel, Martin. “Etude et mise en oeuvre d'un simulateur de noyau avionique ARINC 653 : applications aux cockpits numériques interactifs.” Rapport de projet de fin d'étude, INP-ENSEEIHT, 2013, 42p.

Dearden, Andrex M, and Michael Harrison. “Formalising human error resistance and human error tolerance.” *Proc. of 5th Int. Conf. on Human-Machine Interaction and Artificial Intelligence in Aerospace*. 1995.

Department of the Army. “Failure Modes, Effects and Criticality Analysis (FMECA) for Command, Control, Communications, Computer, Intelligence, Surveillance and Reconnaissance (C4ISR) Facilities.” Technical Manual - TM 5-698-4, United States Army, 2006.

Dessiatnikoff, A., et al. “Securing Integrated Modular Avionics computers.” *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*. 2013. 1-38.

Dix, Alan J. “Formal methods: an introduction to and overview of the use of formal methods within HCI.” Chap. Formal methods: An introduction to and overview of the use of formal methods within HCI in *Perspectives on HCI: diverse approaches*, edited by A. Monk and N. Gilbert (Eds.), 9-43. Academic Press, 1995.

Dix, Alan John. *Formal Methods for Interactive Systems*. Academic Press, 1991.

Dragicevic, Pierre. “Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables.” THESE, Université de Nantes, 2004.

Dubey, Abhishek, Gabor Karsai, and Nagabhushan Mahadevan. “A Component Model for Hard Real-time Systems: CCM with ARINC-653.” *Softw. Pract. Exper.* (John Wiley \& Sons, Inc.) 41, no. 12 (2011): 1517-1550.

Dubey, Abhishek, Gabor Karsai, Robert Kereskenyi, and Nagabhushan Mahadevan. “A Real-Time Component Framework: Experience with CCM and ARINC-653.” *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*. 2010. 143-150.

Dumas, Bruno, Denis Lalanne, Dominique Guinard, Reto Koenig, and Rolf Ingold. “Strengths and Weaknesses of Software Architectures for the Rapid Creation of Tangible and Multimodal Interfaces.” *Proceedings of the 2Nd International Conference on Tangible and Embedded Interaction*. New York, NY, USA: ACM, 2008. 47-54.

EASA. “CS-25.” *Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes*. 2014.

Esteban, Olivier, Stéphane Chatty, and Philippe A. Palanque. “Whizz'Ed: a visual environment for building highly interactive software.” Edited by Knut Nordby, Per H. Helmersen, David J. Gilmore and Svein A. Arnesen. *Human-Computer Interaction, INTERACT '95, IFIP TC13 Interantional Conference on Human-Computer Interaction*. Chapman & Hall, 1995. 121-126.

FAA. “14 CFR Part 25.” *Airworthiness Standards: Transport Category Airplanes*. 2014.

Farenc, Christelle. *Thèse de doctorat: ERGOVAL : une méthode de structuration des règles ergonomiques permettant l'évaluation automatique d'interfaces graphiques*. Toulouse: Université de Toulouse, 1997.

Fayollas, C., C. Martinie, P. Palanque, Y. Deleris, J.-C. Fabre, and D. Navarre. “An Approach for Assessing the Impact of Dependability on Usability: Application to Interactive Cockpits.” *Dependable Computing Conference (EDCC), 2014 Tenth European.* 2014. 198-209.

Fekete, Jean-Daniel, Martin Richard, and Pierre Dragicevic. “Specification and verification of interactors: A tour of Esterel.” *Proceedings of FAHCI* 98 (1998).

Genrich, H.J. “Predicate / Transition Nets.” In *High-level Petri Nets*, edited by Kurt Jensen and Grzegorz Rozenberg, 3-43. Springer Berlin Heidelberg, 1991.

GestureWorks. “GestureML.” *GestureML.* 2014.

Ghezzi, Carlo, Dino Mandrioli, Sandro Morasca, and Mauro Pezze. “A General Way to Put Time in Petri Nets.” *Proceedings of the 5th International Workshop on Software Specification and Design.* New York, NY, USA: ACM, 1989. 60-67.

Gibert, Victor, Mathilde Machin, Jean-Charles Fabre, and Miruna Stoicescu. “Design for Adaptation Fault Tolerance Strategies.” LAAS-RR-12198, LAAS, 2012, 35.

Göransson, Bengt, Jan Gulliksen, and Inger Boivie. “The usability design process – integrating user-centered systems design in the software development process.” *Software Process: Improvement and Practice* (John Wiley & Sons, Ltd.) 8, no. 2 (2003): 111-131.

Gram, Chris, and Gilbert Cockton. *Design Principles for Interactive Software.* Springer, 1996.

Hamilton, Margaret H. “Zero-defect software: The elusive goal: It is theoretically possible but difficult to achieve; logic and interface errors are most common, but errors in user intent may also occur.” *Spectrum, IEEE* 23, no. 3 (1986): 47-53.

Hamon, Arnaud. “Définition d'un langage et d'une méthode pour la description et la spécification d'IHM post-WIMP pour les cockpit interactifs.” PhD Dissertation, Université Toulouse III - Paul Sabatier, 2014.

Hamon, Arnaud, Philippe Palanque, José Luìs Silva, Yannick Deleris, and Eric Barboni. “Formal Description of Multi-touch Interactions.” *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems.* New York, NY, USA: ACM, 2013. 207-216.

Han, Seungjae, Kang G. Shin, and Harold A. Rosenberg. “DOCTOR: an integrated software fault injection environment for distributed real-time systems.” *Computer Performance and Dependability Symposium, 1995. Proceedings., International.* 1995. 204-213.

Hart, Sandra G, and Lowell E Staveland. “Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research.” *Advances in psychology* (Elsevier) 52 (1988): 139-183.

Hix, Deborah, and H Rex Hartson. “Formative evaluation: ensuring usability in user interfaces.” *User interface software.* 1993. 1-30.

IBM. “Common User Access: Advanced Interface Design Guide.” *Document SC26-4582-0.* 1989.

International Standard Organization. “ISO 11898.” *Road vehicles -- Controller area network (CAN).* 2009.

—. “ISO 8807.” *LOTOS - A Formal Description Technique Based on Temporal Ordreing of Observational Behavirour.* 1988.

—. “ISO 9241-11.” *Ergonomic requirements for office work with visual display terminals (VDT) - Part 11 Guidance on Usability.* 1996.

Jensen, Kurt. “Coloured Petri Nets.” *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part I.* London, UK, UK: Springer-Verlag, 1987. 248-299.

Johnson, C.W. “Using Z to support the design of interactive safety-critical systems.” *Software Engineering Journal* 10, no. 2 (1995): 49-60.

Keh, Huan Chao, and T. G. Lewis. “Direct-manipulation User Interface Modeling with High-level Petri Nets.” *Proceedings of the 19th Annual Conference on Computer Science.* New York, NY, USA: ACM, 1991. 487-495.

König, WernerA., Roman Rädle, and Harald Reiterer. “Interactive design of multimodal user interfaces.” *Journal on Multimodal User Interfaces* (Springer-Verlag) 3, no. 3 (2010): 197-213.

Koopman, P., and T. Chakravarty. “Cyclic redundancy code (CRC) polynomial selection for embedded networks.” *Dependable Systems and Networks, 2004 International Conference on.* 2004. 145-154.

Krasner, Glenn E., and Stephen T. Pope. “A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80.” *J. Object Oriented Program.* (SIGS Publications) 1, no. 3 (1988): 26-49.

Ladry, Jean-François. “Une notation et un processus outillé pour le développement de systèmes interactifs multimodaux critiques.” Ph.D. dissertation, Université Toulouse III - Paul Sabatier, 2010.

Laprie, Jean-Claude, et al. *Guide de la sûreté de fonctionnement.* Cépaduès, 1996.

Laprie, Jean-Claude, Jean Arlat, Christian Beounes, and Karama Kanoun. “Definition and analysis of hardware- and software-fault-tolerant architectures.” *Computer* 23, no. 7 (1990): 39-51.

Latoschik, Marc Erich. “Designing Transition Networks for Multimodal VR-Interactions Using a Markup Language.” *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces.* Washington, DC, USA: IEEE Computer Society, 2002. 411--.

Lauer, MMichaël, Jérome Ermont, Frédéric Boniol, and Claire Pagetti. “Worst Case Temporal Consistency in Integrated Modular Avionics Systems.” *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on.* 2011. 212-219.

Lelli, Valéria, Arnaud Blouin, and Baudry Benoit. “Classifying and Qualifying GUI Defects.” *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation.* 2015. 10p.

Limbourg, Quentin. “Multi-Path Development of User Interfaces.” Ph.D. dissertation, Université catholique de Louvain, 2004.

Limbourg, Quentin, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. *USIXML: A Language Supporting Multi-path Development of User Interfaces.* Vol. 3425, in *Engineering Human Computer Interaction and Interactive Systems*, edited by Rémi Bastide, Philippe Palanque and Jörg Roth, 200-220. Springer Berlin Heidelberg, 2005.

Lyons, R.E., and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability.” *IBM Journal of Research and Development* 6, no. 2 (1962): 200-209.

Martinie De Almeida, Célia. “Une approche à base de modèles synergiques pour la prise en compte simultanée de l’utilisabilité, la fiabilité et l’opérabilité des systèmes interactifs critiques.” Ph.D. dissertation, Université Toulouse III - Paul Sabatier, 2011.

Martinie, Célia, Philippe Palanque, and Marco Winckler. *Structuring and Composition Mechanisms to Address Scalability Issues in Task Models.* Vol. 6948, in *Human-Computer Interaction – INTERACT 2011*, edited by Pedro Campos, Nicholas Graham, Joaquim Jorge, Nuno Nunes, Philippe Palanque and Marco Winckler, 589-609. Springer Berlin Heidelberg, 2011.

Martinie, Célia, Philippe Palanque, David Navarre, and Eric Barboni. “A Development Process for Usable Large Scale Interactive Critical Systems: Application to Satellite Ground Segments.” *Proceedings of the 4th International Conference on Human-Centered Software Engineering.* Berlin, Heidelberg: Springer-Verlag, 2012. 72-93.

Martinie, Célia, Philippe Palanque, David Navarre, Marco Winckler, and Erwann Poupart. “Model-based Training: An Approach Supporting Operability of Critical Interactive Systems.”

Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. New York, NY, USA: ACM, 2011. 53-62.

Martinie, Célia, Philippe Palanque, Marco Winckler, and Stéphane Conversy. "DREAMER: A Design Rationale Environment for Argumentation, Modeling and Engineering Requirements." *Proceedings of the 28th ACM International Conference on Design of Communication*. New York, NY, USA: ACM, 2010. 73-80.

Meixner, Gerrit, Marius Orfgen, and Moritz Kümmerling. *Evaluation of User Interface Description Languages for Model-Based User Interface Development in the German Automotive Industry*. Vol. 8004, in *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments*, edited by Masaaki Kurosu, 411-420. Springer Berlin Heidelberg, 2013.

Memon, Atif M., Martah E. Pollack, and Mary Lou Soffa. "Hierarchical GUI test case generation using automated planning." *Software Engineering, IEEE Transactions on* 27, no. 2 (2001): 144-155.

Myers, Brad A., and Mary Beth Rosson. "Survey on User Interface Programming." *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 1992. 195-202.

Navarre, David. "Une technique de description formelle et un environnement pour une modélisation et une exploitation synergiques des tâches et du système." Ph.D. dissertation, Université Toulouse I, 2001.

Navarre, David, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. "ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability." *ACM Transactions on Computer-Human Interaction (TOCHI)*. ACM, 2009.

Navarre, David, Phillippe Palanque, Jean-François Ladry, and Sandra Basnyat. "An Architecture and a Formal Description Technique for the Design and Implementation of Reconfigurable User Interfaces." *Design Specification and Verification of Interactive Systems, DSVIS'08*. Kingston, Ontario, Canada: Springer-Verlag Berlin, 2008.

Nielsen, Jakob. *Discount Usability: 20 Years*. 2009. http://www.nngroup.com/articles/discount-usability-20-years/ (accessed 04 29, 2015).

—. *Why You Only Need to Test with 5 Users*. 2000. http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (accessed 04 29, 2015).

Nielsen, Jakob, and Robert L. Mack, . *Usability inspection methods*. New York: Wiley, 1994.

Nigay, Laurence, and Joëlle Coutaz. "A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion." *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 1993. 172-178.

Norman, Donald A. "The Design of Everyday Things." *(Originally published: The psychology of everyday things)*. New York: Basic Books, 1988.

Normand, Eugène. "Single event upset at ground level." *Nuclear Science, IEEE Transactions on* 43, no. 6 (1996a): 2742-2750.

Normand, Eugène. "Single-event effects in avionics." *Nuclear Science, IEEE Transactions on* 43, no. 2 (1996b): 461-474.

Oney, Stephen, Brad Myers, and Joel Brandt. "ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States." *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2012. 229-238.

Palanque, Philippe. "Modélisation par Objets Cooperatifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur." Ph.D. dissertation, Université de Toulouse 1, 1992.

Palanque, Philippe, and Bastide Rémi. “Verification of an Interactive Software by Analysis of its Formal Specification.” *INTERACT'95 conference.* 1995. 6p.

Palanque, Philippe, and Rémi Bastide. “A Formalism for Reliable User Interfaces.” *Workshop on Software Engineering and Human Computer Interaction associated with the 16th IEEE ICSE conference.* Sorrento, Italy, 1994.

Palanque, Philippe, Eric Barboni, Célia Martinie, David Navarre, and Marco Winckler. “A Model-based Approach for Supporting Engineering Usability Evaluation of Interaction Techniques.” *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems.* New York, NY, USA: ACM, 2011. 21-30.

Palanque, Philippe, Regina Bernhaupt, David Navarre, Mourad Ould, and Marco Winckler. “Supporting Usability Evaluation of Multimodal Man-Machine Interfaces for Space Ground Segment Applications Using Petri nets Based Formal Specification.” Edited by American Institute of Aeronautics and Astronautics. *Proceedings of the Conference on Space Operations 2006 (SpaceOps).* 2006.

Palanque, Philippe, Rémi Bastide, and Louis Dourte. “Contextual Help for Free With Formal Dialogue Design.” *Proc. of the Fifth International Conference on Human-Computer Interaction (HCI International '93).* 1993.

Palmer, Everett. “Oops, it didn’t arm.” Edited by R. S. Jensen and L. A. Rakovan. *Proceedings of the Eighth International Symposium on Aviation Psychology.* Columbus, OH, USA, 1995. 227–232.

Parnas, David L. “On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System.” *Proceedings of the 1969 24th National Conference.* New York, NY, USA: ACM, 1969. 379-385.

Pascoal, E., J. Rufino, T. Schoofs, and J. Windsor. “AMOBA - ARINC 653 Simulator for Modular Space Based Applications.” 2008.

Paterno, Fabio, and Menica Mezzanotte. “Analysing MATIS by Interactors and ACTL.” *Amodeus Project Document: System Modelling/WP36*, 1994.

Petri, C.A. *Kommunikation mit Automaten.* Rhein.-Westfäl. Inst. f. Instrumentelle Mathematik an der Univ. Bonn, 1962.

Pfaff, G. E., ed. *User Interface Management Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1985.

Pintard, L., J.-C. Fabre, M. Leeman, K. Kanoun, and M. Roy. “From Safety Analyses to Experimental Validation of Automotive Embedded Systems.” *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on.* 2014. 125-134.

Pintard, Ludovic. “From Safety Analysis to Experimental Validation by Fault Injection - Case of Automotive Embedded Systems.” Ph.D. dissertation, Institut National Polytechnique de Toulouse, 2015.

Pnueli, A. *Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends.* Vol. 224, in *Current Trends in Concurrency Lecture Notes in Computer Science*, 510-584. Springer, 1986.

Prisaznuk, Paul J. “ARINC 653 role in Integrated Modular Avionics (IMA).” *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th.* 2008. 1.E.5-1-1.E.5-10.

—. “Integrated modular avionics.” *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National.* 1992. 39-45 vol.1.

Randell, Brian. “System structure for software fault tolerance.” *Software Engineering, IEEE Transactions on* SE-1, no. 2 (June 1975): 220-232.

Reason, James. *Human Error.* Cambridge University Press, 1990.

Regis, Didier, Guillaume Hubert, Frank Bayle, and Marc Gatti. “IC components reliability concerns for avionics end-users.” *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd.* 2013. 2C2-1-2C2-9.

Rieder, Rafael, Alberto Barbosa Raposo, and Marcio Sarroglia Pinho. “A Methodology to Specify Three-dimensional Interaction Using Petri Nets.” *J. Vis. Lang. Comput.* (Academic Press, Inc.) 21, no. 3 (#jun# 2010): 136-156.

RTCA, and EUROCAE. “DO-178C/ED-12C.” *Software Considerations in Airborne Systems and Equipment Certification.* 2011a.

—. “DO-254/ED-80.” *Design Assurance Guidance for Airborne Electronic Hardware.* Radio Technical Commission for Aeronautics (RTCA), 2000.

—. “DO-331/ED-218.” *Model-Based Development and Verification Supplement to DO-178C and DO-278A.* 2011b.

—. “DO-333/ED-216.” *Formal Methods Supplement to DO-178C and DO-278A.* 2011c.

Rushby, John. “Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance.” NASA Contractor Report, NASA Langley Research Center, 1999.

SAE International. “ARP 4754.” *Certification Considerations for Highly-Integrated Or Complex Aircraft.* 2010.

—. “ARP 4761.” *Guidelines and Methods for Conducting the Safety Assessment Process on Civil.* 2004.

—. “AS 5506.” *AADL - Architecture Analysis and Design Language.* 2012.

Santos, Sergio, Jose Rufino, Tobias Schoofs, Cassia Tatibana, and James Windsor. “A portable ARINC 653 standard interface.” *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th.* 2008. 1.E.2-1-1.E.2-7.

Schoofs, Tobias, Sergio Santos, Cassia Tatibana, and Jose Anjos. “An integrated modular avionics development environment.” *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th.* 2009. 1.A.2-1-1.A.2-9.

Silva, José Luis, Camille Fayollas, Arnaud Hamon, Philippe Palanque, Celia Martinie De Almeida, and Eric Barboni. “Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification (regular paper).” *International Workshop on Formal Methods for Interactive Systems (FMIS), London, 24/06/2013.* http://www.elsevier.com/: Elsevier, 2013.

Smith, Shamus, and David Duke. “Using CSP to specify interaction in virtual environments.” *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS* (UNIVERSITY OF YORK), 1999.

Sommerville, Ian. *Software Engineering.* 9e. Addison-Wesley, 2011.

Souchon, Nathalie, and Jean Vanderdonckt. *A Review of XML-compliant User Interface Description Languages.* Vol. 2844, in *Interactive Systems. Design, Specification, and Verification*, edited by JoaquimA. Jorge, Nuno Jardim Nunes and João Falcão e Cunha, 377-391. Springer Berlin Heidelberg, 2003.

Stavely, Allan M. *Toward Zero-Defect Programming.* 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

Stoicescu, Miruna. “Architecting Resilient Computing Systems: a Component-Based Approach.” Ph.D. dissertation, Université de Toulouse - Institut National Polytechnique de Toulouse, 2013.

Stoicescu, Miruna, Jean-Charles Fabre, and Matthieu Roy. “Architecting Resilient Computing Systems: Overall Approach and Open Issues.” *Proceedings of the Third International Conference on Software Engineering for Resilient Systems.* Berlin, Heidelberg: Springer-Verlag, 2011. 48-62.

Storey, Neil R. *Safety Critical Computer Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

Sullivan, M., and R. Chillarege. “Software defects and their impact on system availability-a study of field failures in operating systems.” *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium.* 1991. 2-9.

Svenningsson, Rickard. “Model-Implemented Fault Injection for Robustness Assessment.” Licienciate thesis, KTH Royal Institue of Technology, 2011, 51p.

Svenningsson, Rickard, Henrik Eriksson, Johny Vinter, and Martin Törngren. “Model-Implemented Fault Injection for Hardware Fault Simulation.” *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on.* 2010. 31-36.

Taber, A., and E. Normand. “Single event upset in avionics.” *Nuclear Science, IEEE Transactions on* 40, no. 2 (1993): 120-126.

Tankeu-Choitat, A., D. Navarre, P. Palanque, Y. Deleris, J-C Fabre, and C. Fayollas. “Self-Checking Components for Dependable Interactive Cockpits Using Formal Description Techniques.” *Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing.* Washington, DC, USA: IEEE Computer Society, 2011. 164-173.

Tankeu-Choitat, Adrienne. “Approches outillées pour le développement des systèmes interactifs intégrant les aspects sûreté de fonctionnement et utilisabilité.” Ph.D. dissertation, Université Toulouse III - Paul Sabatier, 2011.

Thimbleby, Harold. *User-Centered Methods Are Insufficient for Safety Critical Systems.* Vol. 4799, in *HCI and Usability for Medicine and Health Care*, edited by Andreas Holzinger, 1-20. Springer Berlin Heidelberg, 2007.

Thimbleby, Harold, and Andy Gimblett. “Dependable keyed data entry for interactive systems.” *Proceedings of the 4th International Workshop on Formal Methods for Interactive Systems (FMIS'2011).* 2011.

Traverse, Pascal, Isabelle Lacaze, and Jean Souyris. “Airbus fly-by-wire - A total approach to dependability.” *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France.* 2004. 191-212.

van Dam, Andries. “Post-WIMP User Interfaces.” *Commun. ACM* (ACM) 40, no. 2 (1997): 63-67.

Van Schooten, Boris, Olaf Donk, and Job Zwiers. “Modelling interaction in virtual environments using process algebra.” *TWLT15: Interactions in Virtual Worlds* (Citeseer), 1999.

Vinter, J., L. Bromander, P. Raistrick, and H. Edler. “FISCADE - A Fault Injection Tool for SCADE Models.” *Automotive Electronics, 2007 3rd Institution of Engineering and Technology Conference on.* 2007. 1-9.

Wang, Kang, Shaoping Wang, and Jian Shi. “Integrated reliability theory and evaluation methodology of AFDX.” *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on.* 2012. 657-662.

Wasserman, Anthony I. “User Software Engineering and the Design of Interactive Systems.” *Proceedings of the 5th International Conference on Software Engineering.* Piscataway, NJ, USA: IEEE Press, 1981. 387-393.

Wegner, Peter. “Why Interaction is More Powerful Than Algorithms.” *Commun. ACM* (ACM) 40, no. 5 (1997): 80-91.

Willans, James S., and Michael D. Harrison. “Prototyping Pre-implementation Designs of Virtual Environment Behaviour.” *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction.* London, UK, UK: Springer-Verlag, 2001. 91-108.

Wright, Nicholas, Andrew S. Patrick, and Robert Biddle. “Do You See Your Password?: Applying Recognition to Textual Passwords.” *Proceedings of the Eighth Symposium on Usable Privacy and Security.* New York, NY, USA: ACM, 2012. 8:1--8:14.

Yeh, Ying .C. (Bob). “Triple-triple redundant 777 primary flight computer.” *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*. 1996. 293-307 vol.1.

Yeh, Ying C. (Bob). “Design Considerations in Boeing 777 Fly-By-Wire Computers.” *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*. 1998. 64-73.

Ziegler, J. F., and W. A. Lanford. “Effect of Cosmic Rays on Computer Memories.” *Science* 206, no. 4420 (1979): 776-788.

Ziegler, J.F., et al. “IBM experiments in soft fails in computer electronics (1978-1994).” *IBM Journal of Research and Development* 40, no. 1 (1996): 3-18.

RÉSUMÉ

Depuis l'introduction au début des années 2000 du standard ARINC 661 (définissant les interfaces graphiques dans les cockpits), les avions modernes, tels que l'A380, l'A350 ou le B787, intègrent des systèmes interactifs permettant à l'équipage d'interagir avec des applications interactives. Ces applications sont affichées sur des écrans à travers l'utilisation d'un dispositif similaire à un clavier et une souris. Pour des raisons d'exigences de sûreté de fonctionnement, l'utilisation de ces systèmes est limitée, à l'heure actuelle, à la commande et au contrôle de fonctions avioniques non critiques. Cependant, l'utilisation de ces systèmes dans les cockpits d'avions civils apporte de nombreux avantages (tels qu'une amélioration de l'évolutivité du cockpit) qui amènent les industriels à chercher comment l'étendre à la commande et le contrôle de systèmes avioniques critiques. Dans cette optique, nous proposons une approche duale et homogène de prévention et de tolérance aux fautes pour concevoir et développer des systèmes interactifs tolérants aux fautes. Celle-ci repose, dans un premier temps, sur une approche à base de modèles permettant de décrire de manière complète et non ambiguë les composants logiciels des systèmes interactifs et de prévenir les fautes logicielles de développement. Dans un second temps, elle repose sur une approche de tolérance aux fautes naturelles et certaines fautes logicielles résiduelles en opération, grâce à la mise en œuvre d'une solution architecturale fondée sur le principe des composants autotestables. Les contributions de la thèse sont illustrées sur une étude de cas de taille industrielle : une application interactive inspirée du système de commande et contrôle de l'autopilote de l'A380.

Mots-clefs : Systèmes interactifs critiques, Sûreté de fonctionnement, Architecture logicielle, Approche à base de modèles, Tolérance aux fautes, Cockpits avioniques

ABSTRACT

Since the introduction of the ARINC 661 standard (that defines graphical interfaces in the cockpits) in the early 2000, modern aircrafts such as the A380, the A350 or the B787 possess interactive systems. The crew interacts, through physical devices similar to keyboard and mouse, with interactive applications displayed on screens. For dependability reasons, only non-critical avionics systems are managed using such interactive systems. However, their use brings several advantages (such as a better upgradability), leading aircraft manufacturers to generalize the use of such interactive systems to the management of critical avionics functions. To reach this goal, we propose a dual and homogeneous fault prevention and fault tolerance approach. Firstly, we propose a model-based approach to describe in a complete and unambiguous way interactive software components to prevent as much as possible development software faults. Secondly, we propose a fault tolerant approach to deal with operational natural faults and some residual software faults. This is achieved through the implementation of a fault tolerant architecture based on the principle of self-checking components. Our approach is illustrated on a real size case study: an interactive application based on the command and control system of the A380 autopilot.

Keywords: Interactive Critical Systems, Dependability, Software Architecture, Model-Based Approach, Fault-tolerance, Aircraft cockpits